# IMPROMPTU: SOFTWARE FRAMEWORK FOR SELF-HEALING MIDDLEWARE SERVICES

**Ella Grishikashvili Pereira**

A thesis submitted in partial fulfilment of the requirements of Liverpool

John Moores University for the degree of

Doctor of Philosophy

May 2006

*To My Mother*

# Abstract

As a result of recent advancements in the fields of computer networks, software and hardware technology, networked systems and appliances have improved immensely in terms of performance, reliability and cost of ownership. This undoubtedly has been a major catalyst for its widespread adoption and interweaving in today's socio-economic fabric (everyday life). Whilst, such a development has created many commercial opportunities with slogans including; "anywhere anytime" and "always on", it has engendered a range of technical challenges including: how to ensure systems' lifetime dependability and high-availability.

This has driven research and development to exploring new generations of networked software systems' design, management and maintenance along a number of directions including: component-based and service-oriented models. Where software is no longer developed as large monolithic systems, but rather they are assembled from Commercial Off The Shelf (COTS) or Services Of The Server (SOTS).

Whilst, COTS and SOTS approaches have improved software design, maintenance and evolution, however, they have engendered additional challenges – as distributed applications are notoriously difficult to develop and manage due to their inherent dynamics, and heterogeneity of their implementation, topology, deployment and network requirements. Middleware technology has come to the rescue by facilitating the development and interoperation of distributed applications. Though, much more research is required, for instance, to support *On-demand* self-assembly and healing of software application, which is the main focus of this work, that is, to investigate the fundamental requirements for a software framework and associated middleware services to develop on-demand application services.

This work makes a number of contributions towards a better understanding of software self-healing requirements for autonomic distributed software engineering, including: on-demand service assembly and delivery.

# Acknowledgements

I would like to thank all those people who have supported me throughout this PhD. Firstly I wish to express my gratitude to my supervisor Professor A. Taleb-Bendiab for his valuable guidance, advice, significant feedback and criticism. I also would like to thank to the director of the school Professor Madjid Merabti for giving me the opportunity to come to Liverpool to carry out my PhD studies.

Many thanks to my second supervisor Mr. Andy Laws and colleague Mr. Denis Reilly for their assistance, useful guidance and numerous discussions that clarified many aspects of this work. Also, my thanks to all colleagues, academic staff, administration staff, technicians and research students in the School of Computing and Mathematical Science for creating a friendly and supportive environment during these years.

My special thanks and gratitude to my friends Brian and Jeni Fogg for their warmth, special friendship and encouragement since I first came to England.

I would like to express many thanks to my husband Rubem for his patience, continuous help and support and to my son Nicholas Alexander for being an indirect inspiration to complete this work.

Last but not least I would like to thank all my family for their strong support and encouragement.

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation: High-availability Concerns of Distributed Applications

As a result of recent advancements in the fields of computer networks, software and hardware technology, networked systems and appliances have improved immensely in terms of performance, reliability and cost of ownership. This undoubtedly has been a major catalyst for their widespread adoption and interweaving in today's socio-economic fabric (everyday life).

While such a development has created many commercial opportunities with slogans including; "anywhere anytime" and "always on", it has also engendered a range of technical challenges including: how to ensure systems' lifetime dependability, high-availability and assurance. This has driven research and development to explore new generations of networked software systems' design, management and maintenance along a number of directions including: component-based and service-oriented models [3]. Here, software is no longer developed as large monolithic systems but rather from Commercial Off The Shelf components (COTS) [4] or Services Of The Server (SOTS). Such application services make use of infrastructure software that has also been decomposed into discrete system services. These discrete services can be deployed across any number of physical machines that are interconnected. By reassembling a few services into a new configuration, a user can create a new service.

Whilst, COTS and SOTS approaches have improved software design, maintenance and evolution, however, they add additional challenges – as distributed applications are notoriously difficult to develop and manage due to their inherent dynamics, and heterogeneity of their implementation, topology, deployment and network

requirements. Middleware technology has come to the rescue by facilitating the development and interoperation of distributed applications. Until recently however, little attention has focused on combining an assembly concept in conjunction with middleware technologies to understand dynamic behaviour and assist with the runtime management of distributed applications.

## 1.2 Challenges: On-Demand Service Delivery and Management

One of the challenges in distributed systems development is the concept of on-demand service delivery and management. It is a foundation for modular, flexible, and automated access to digital assets, including computing resources, from virtually anywhere. The vision of on-demand services is a framework, encompassing networked services such as: services registry, lookup, and discovery, along with more advanced capabilities, such as virtualised containers (storages), composite services (created by combining separate services) and behaviours such as self-adaptation and self-healing.

It is challenging to create a system that gives the user the ability anytime, from anywhere to access over the local or global network services that are dynamically discoverable, reusable and re-configurable into flexible, interoperable, innovative applications. The creation of this kind of system involves a range of technical issues to be addressed and requires the development of:

- Reference models: including software design patterns, design models and architectures to aid developers in the design, development and management of self-managing 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 and requirements.
- Mechanisms: to be used for rapid assembly of distributed components to form a trusted service that can still be analysed to determine that it meets its requirements. To this end, other facilities and utilities need to be developed, including:

- How to access and reason about functionalities that distributed services offer: the distributed services that run in different locations and are developed by different individuals.

- How to create a common understanding between these service providers and clients.

- How to teach a system to find and invoke the service the user requires.

- How to teach a system to monitor the behaviour of each composition and repair it in case of failure.

- Experimental insight: demonstrating that this type of mechanism enables computers, software components and devices to plug together quickly to form impromptu, networked applications assembled with less human intervention and moreover to make them self-organising and self-healing.

## 1.3 Research Hypothesis

Influenced by the service-oriented programming [11, 12], and the IBM autonomic computing blueprint [13], this work proposes that autonomic computing capabilities including: self-healing, self-optimisation, self-configuration and self-protection to be provided as middleware services.

Hence, with specific focus on self-healing capability only, the hypothesis for this study can be formulated as:

> *Practical self-healing properties can be incorporated in distributed services-based applications through the extension of existing middleware technologies.*

## 1.4 Approach

The work described in this thesis aims to investigate the fundamental requirements of a software framework and associated middleware services to develop on-demand

application services. The software framework is intended to enable user to access over the local or global network, services that are dynamically discoverable, reusable and combinable into flexible self-adaptive applications.

To build such a system requires encompassing networked services, such as registry, lookup and discovery along with more advanced services, including:

- *The assembly service*: containing the service description, service configuration and service execution functionalities.

- *The system manager service*: containing monitoring and self-healing (recovery and reconfiguration) capabilities.

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

- Advanced software engineering: using middleware services to bridge the gap between the network layer and the application layer [5]. The services such as lookup, registry or discovery. The remote event notification concept [6] to enhance the communication between the system and services and remote method invocation techniques for service invocation [7].

- Self-healing systems: using proposed models, requirements and theories to enable software to use real-time monitoring, diagnosis, repair and control [8].

In addition, this work follows an experimental research approach aimed at the design, build and test of a framework for on-demand service delivery software with self-healing behaviour.

The well-established Viable Systems Model provides a design blueprint and system taxonomy to guide the specification and design of the intended autonomic middleware services and associated reference architectural model.

Furthermore, it this work aims to document a pattern language for on-demand service delivery and management software by testing the existing framework with different application scenarios.

## 1.5 Contributions

This work makes a number of contributions towards a better understanding of self-healing software requirements for autonomic distributed software engineering. Such contributions are summarised below:

- *A software architectural model*: motivated by and grounded in a range of current research on distributed middleware, service-oriented architectures, service on demand concepts and part of autonomic computing for its support of self-healing and self-awareness. In particular, this work contributes to the development of an architectural model for a middleware-based on-demand self- servicing software framework. The framework described in this thesis is developed based on a proposed architectural model and unites a number of components including: service description, discovery, invocation, configuration, monitoring and response to events, such as failure of the service.

- *A mechanism for assembling distributed services regardless of their location* - is a representation of how a single functionality provided by one service can be combined with the functionalities provided by another service to form a required composite service. As a part of the *Impromptu* framework this mechanism contains a number of novel components including:

    - The S*ervice Description Model:* providing a meta description of a given software service. The development of such a model was motivated by the lack of uniform description of software services.

    - The *Service Invocation Model* - based on the above service meta model, it facilitates the invocation of a discovered, distributed service. The service invocation model is responsible for parsing the

service description document, then via the reflection API, the service is assembled and invoked.

- The *Operational Service* – is a novel element of the framework referred to here as a virtual container. This was introduced to provide support for application services development and provides a virtual environment where the services are grouped and executed according to users' requirements. Such an environment helps the monitoring and management of applications, so that the system knows where to look for a particular service assemblage.

- *An Adaptation model* that is responsible for providing self-healing capabilities. For instance, the system responds to a detected service failure by searching for an alternative service in a local or global network. System self-healing is performed when the new service is found and invoked.

- *A Design Pattern Language* focused on on-demand software service assembly and delivery to support software self-healing behaviour. The design pattern language includes patterns for *ad-hoc* service discovery, invocation and monitoring.

## 1.5 Scope

This research proposed a new software framework and associated autonomic middleware services that enable the development of distributed applications, on demand, according to user's requirements. Furthermore, the system is able to perform runtime self-healing when a failure of one or more components has been detected. In particular, this work focuses on:

- The development of a generic model for distributed services to be discovered, invoked and monitored throughout their lifetime. A self-healing process is triggered if and when a system failure has been detected.

- The development of a software framework that offers a number of middleware service that perform the activities described above (discovery, invocation monitoring, recovery, etc.)

- The development of a service description model that helps the client to obtain the service signatures described in the service interfaces by the service providers.
- The development of a design pattern language for On-demand Service Delivery and Assembly of software.

Furthermore, in a constantly changing environment where services come and go, using our software model the developers will be able to search for, discover and bind to a service matching their required service profile. If no match is found or available the framework enables the selection of the closest match, and/or activating a required service from the code base[1]. In addition, the Impromptu framework enables the generation and publication of new application services for other users and applications to use.

The main building blocks of the Impromptu architecture are shown in Figure 1.1.

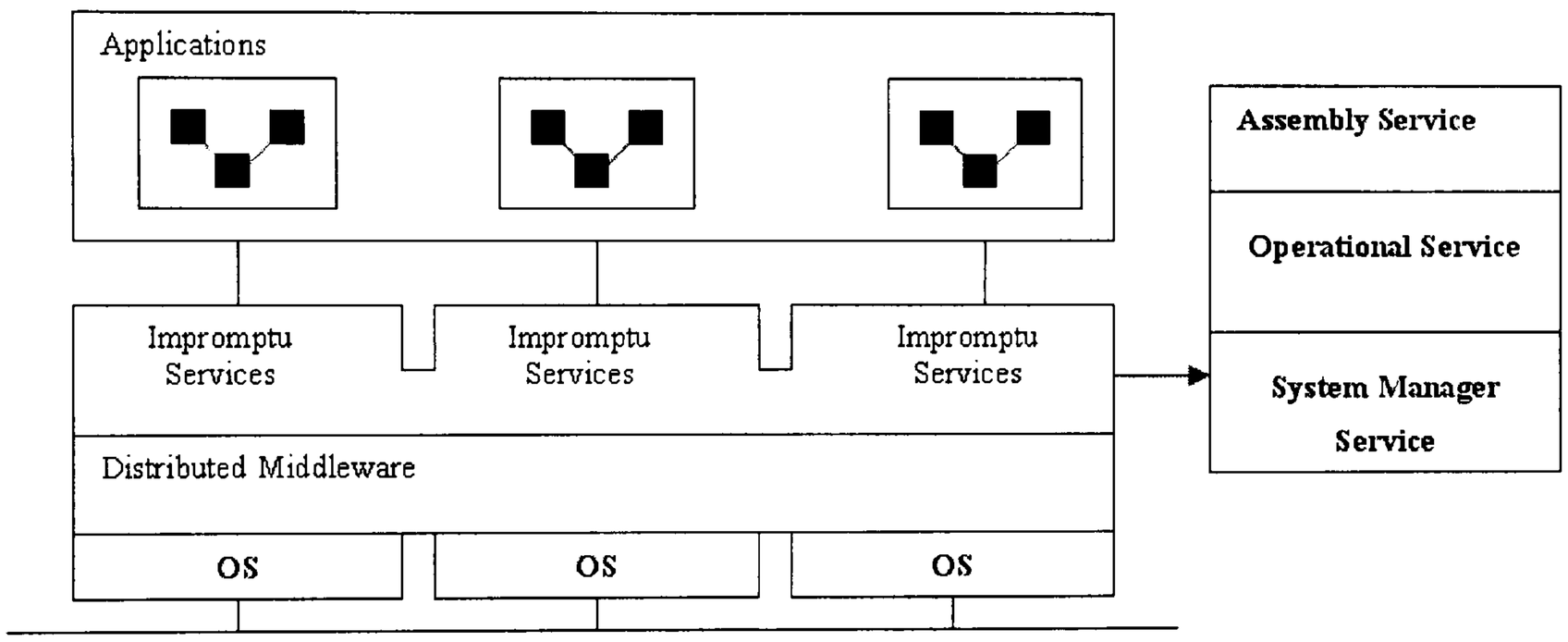



**Figure1.1: Impromptu Components**

The approach for developing the generic model described in this thesis has been tested in two different example applications. One example application was developed for a home appliances scenario in order to test the simple services. And the other example application is based on more complex software. In both cases the applications were tested.

## 1.6 Thesis Structure

[1] A full description of the software development model adopted for the proposed On-demand Service Assembly Delivery (OSAD) is given in Chapter 4

This thesis consists of nine chapters and is organized as follow:

In this chapter we have discussed the main motivations for the work, challenges, contributions and thesis outline.

Chapter 2 introduces the relevant background theories, principles and technologies that are used or considered important for the development of the proposed work. The survey covers the basic concepts and principles of distributed systems including: Distributed Middleware technologies, Service-oriented and Component-oriented Developments. We also review Autonomic Computing and its architectural concepts.

Chapter 3 provides a literature review of related work covering a range of fields, including: Self-healing Systems, Service-oriented architectures and frameworks as well as Component-based frameworks. At the end of this chapter we list the main requirements for the proposed framework development.

Chapter 4 describes the overall design of the proposed On-demand Service Assembly and Delivery (OSAD) model and consequent framework services as well as the concept of self-healing in the OSAD model.

Chapter 5 presents the architecture of the model, describing where the framework middleware services fit.

Chapter 6 provides a prototypical implementation of the model based on the Java programming language and Java-based Jini middleware technology. This chapter also contains the description of two illustrative example applications, the first based on one of the well-known connected homes scenarios and the second, a Software services example.

Chapter 7 documents a pattern language for distributed self-healing applications development.

Chapter 8 describes an evaluation of our work, giving the assumptions of the previous chapters and using example applications such as sorting algorithm services and the home appliances application.

Chapter 9 presents a summary, concluding remarks and proposed future work.

# Chapter 2

# Background

## 2.1 Introduction

Distributed applications are difficult to develop and manage due to their inherent dynamics, the heterogeneity of component technologies and the possibility of different network protocols. Distributed component-based software applications often consist of a collection of software components that communicate via distributed middleware. The distributed middleware, or simply middleware plays a crucial role by providing APIs and support functions to bridge the gap between the network operating system and distributed components and services.

This chapter provides an overview of existing supporting technologies for distributed self-adaptive applications development. This starts with a brief description of the basic concepts and principles of distributed systems (Sec. 2.2), and distributed object-oriented programming and middleware including; CORBA technology, DCOM (Sec. 2.3.1) and Jini technology (Sec. 2.3.2). The chapter concludes with an overview of the principles and mechanisms to support the design, development and management of autonomic computing (Sec. 2.6).

## 2.2. Distributed Systems: basic concepts and principles

Much research work related to the design and development of distributed systems and their underlying principles has been performed. The development of distributed systems followed the emergence of high-speed, local area computer networks at the beginning of the 1970s. More recently, the availability of high-performance personal computers, workstations and server computers has resulted in a major shift towards distributed systems and away from centralized and multi-user computers. This trend has been accelerated by the development of distributed system software, designed to support the development of distributed applications.

Different definitions of distributed systems have been given in the literature. One defines it as:

> *"... a collection of independent computers that appears to its users as a single coherent system."* [9]

Another as:

> *"...one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages"* [10].

The definitions may be as different as the individuals that define them, but the goal is the same: to make it easy for users to access remote resources and to share them with other users in a controlled fashion. Resources can be anything, but typical examples include computers, printers, storage facilities, data, files, Web pages and applications to name but a few. The key characteristics of distributed systems can be summarised as follow [11]:

- Heterogeneity: distributed systems should be constructed from a variety of different networks, operating systems, computer hardware and programming languages.
- Transparency: is an ability for the system to present itself to users and applications as if it was only a single computer system. Transparency deals with hiding the fact that its processes and resources are physically distributed across multiple computers.
- Openness: distributed systems should be extensible. An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.
- Scalability: a system can be scalable with respect to its size, meaning that we can add more users and resources to the system, or it can be geographically scalable in which the users and resources may lie far apart.
- Fault-handling: any process, computer or network may fail independently of the others. Therefore, each component needs to be aware of the possible ways in which the components it depends on may fail and be designed to deal with each of those failures appropriately.

- Concurrency: the presence of multiple users in a distributed system is a source of concurrent requests for its resources. Each resource must be designed to be safe in a concurrent environment.

There are different types of distributed system. A *distributed operating system* manages the hardware of tightly coupled computer systems, which include multiprocessors and homogeneous multi-computers. These distributed systems are good at providing a single-system view, but do not support autonomous computers. A *network operating system*, on the other hand, is good at connecting different computers, each with their own operating system, so that users can easily make use of each node's local services. However, network operating systems do not offer a single-system view the way that distributed operating system do. Neither a distributed operating system nor a network operating system can really support the development of a distributed system that has the best of both worlds: the scalability and openness of network operating systems and the transparency of distributed operating systems. The solution was found in an additional layer of software that is used in a network operating system to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency. Modern distributed systems are constructed by means of such an additional layer of what is called Middleware [12]: the layer that is placed between network operating systems and the application layer. Middleware with real case examples of existing distributed systems constructed, as middleware will be described in detail in Section 2.3.

An important issue with a distributed system is its organization. The most widely applied model is that of *client* processes requesting services at *server* processes (Fig. 2.1). A client sends a message to the server and waits until the latter replies. This client-server interaction, also known as request-reply behaviour, is strongly related to traditional programming, in which services are implemented as procedures in separate modules. A further refinement is often made by distinguishing a user-interface level, a processing level, and a data level.

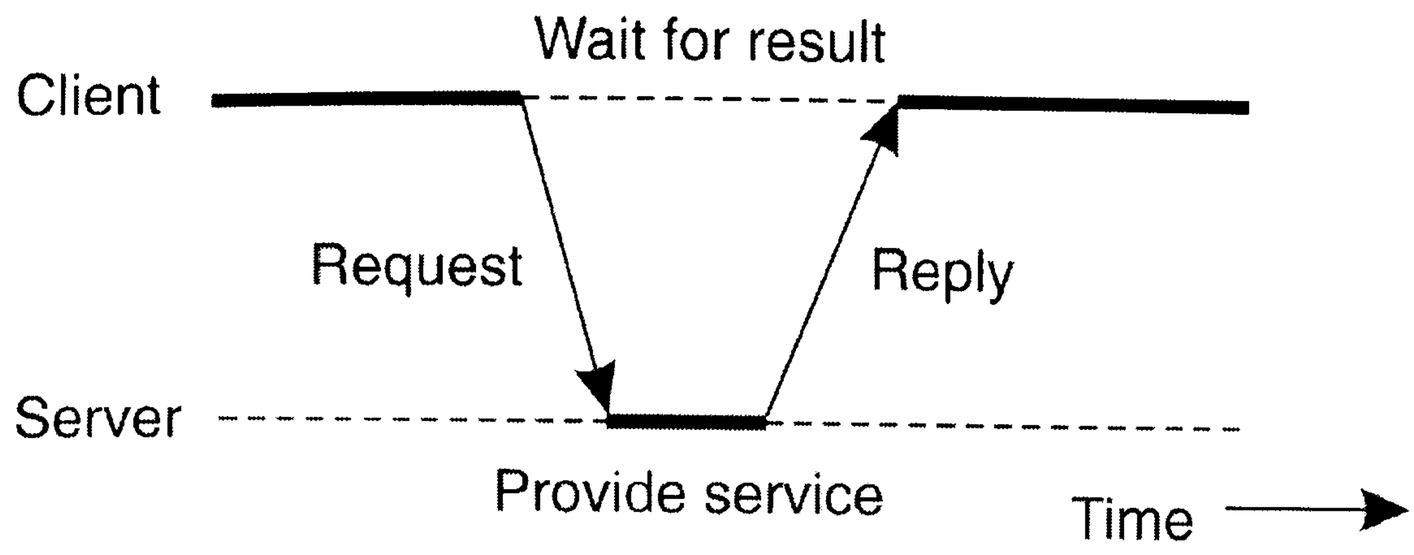



**Figure 2.1: A client-Service interaction [12]**

The user-interface level contains all that is necessary to directly interface with the user. The processing level contains the applications and the data level contains the actual data that is being acted on.

Distinguishing three logical levels suggests a number of possibilities for physically distributing a client-server application across several machines. The consequence of this division was the creation of multi-tiered client-server architectures [9]. The different tiers correspond directly with the logical organization of applications. There is *vertical distribution* that can be characterized as: the server is generally responsible for the data level, a processing level and the user-interface level is implemented at the client side. The processing level can be implemented at the client, server, or split between the two.

For modern distributed systems, this vertical organization of client-server applications is not sufficient to build large-scale systems. What is needed is a *horizontal distribution* by which clients and servers are physically distributed across multiple computers. This type of distribution has been successfully implemented in the World Wide Web (App. A).

In a *peer-to-peer distribution* with one user seeking contact with another, both can launch the same application start a session. A third client may contact either of them, and subsequently also launch the same application software. One of the essential issues, after organization, is the communication between processes in a distributed system. In traditional network applications, communication is often based on low-level message-passing primitives. In modern distributed systems, the widely used communication models are: Remote Procedure Call (RPC), Remote Method Invocation (RMI), Message-Oriented Middleware (MOM), and streams.

The essence of an RPC is that a service is implemented by means of a procedure, of which the body is executed at a server. When the client calls the procedure, the client side implementation, called a stub, takes care of wrapping the parameter values into a message and sending that to the server. The server calls the actual procedure and returns the results, again in a message. The client's stub extracts the result values from the return message and passes it back to the calling client application.

A Remote Method Invocation (RMI) is similar to an RPC in terms of parameter passing. Though, an essential difference between them is that RMI supports system wide object references to be passed as parameters. RPC and RMI offer synchronous communication facilities, by which a client is blocked until the server has sent a reply. In this sense, the message-oriented models (that underpin MOM) offer more convenient inter process communications as they offer asynchronous communication; that is, the sender is allowed to continue immediately after the message has been submitted for transmission. MOMs are used to assist the integration of (widely dispersed) collections of databases into large-scale information systems[2].

## 2.3 Middleware Based Distributed Systems

To support heterogeneous computers and networks while offering a single-system view, distributed systems are often conceptually organized into a layered topology of software, which are logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of network operating systems (Fig. 2.2). Such a layer is often referred to as middleware.

Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system, as shown in Figure 2.2. In doing so, it provides a higher-level building block for programmers than Application Programming Interfaces (APIs) such as sockets that are provided by the operating system.

---

[2] Streaming is a completely different type of communication and is used mainly for video and audio streaming.

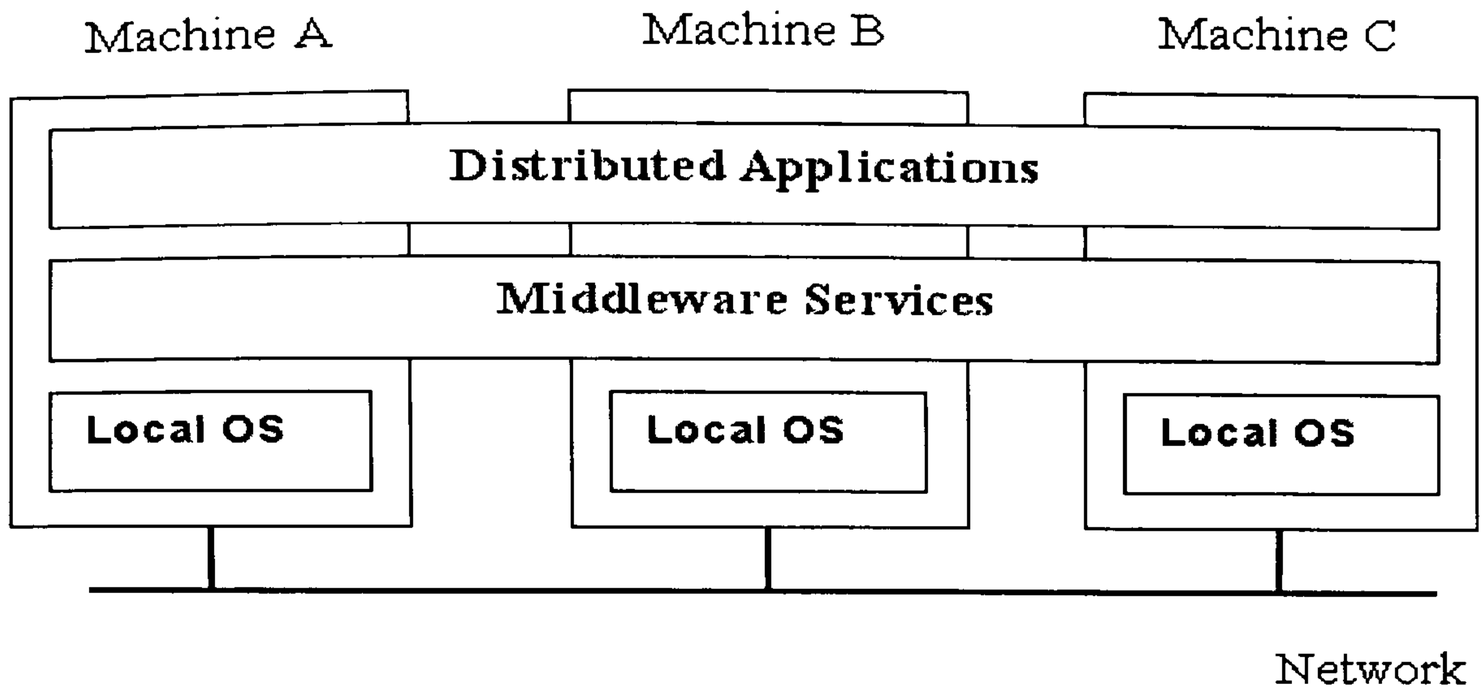



**Figure 2.2: A Distributed System organized as a middleware [12]**

Middleware frameworks are designed to mask systems' heterogeneity providing distributed systems developers with convenient programming models for distributed object-oriented programming including support for: location transparency, concurrency, replication, failures and mobility.

## 2.3.1 Categories of Middleware

Many types of middleware now exist, which can be grouped according to the programming abstractions they provide:

- **Distributed Tuple Space Middleware:** this is based on a distributed relational database offering the abstractions of distributed tuples. Its Structured Query Language (SQL) [13] allows programmers to manipulate sets of these tuples (a database) with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus. A good example of this type of middleware is the Linda framework [14], which offers a distributed tuple abstraction called Tuple Space (TS) [15].

- **Remote Procedure Call Middleware:** this extends the procedure call interface to offer the abstraction of being able to invoke a procedure whose body is across a network. Remote Procedure Call systems are usually synchronous, and thus offer no potential for parallelism without using multiple threads, and they typically have limited exception-handling facilities.

- **Distributed Object Middleware:** this requires that each object implements an interface that hides all the internal details of the object from the users (remote object). Thus the only thing that a process sees of an object is its

interface, that is, the set of method signatures that the object implements. The best known example of such a middleware are; Common Object Request Broker Architecture (CORBA) [16], Jini [5] and DCOM [17}. CORBA is a standard for distributed object computing proposed and developed by the Object Management Group (OMG) [OMG, #24]. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching. CORBA offers heterogeneity across programming languages and vendor implementations. Its standards are publicly available and well defined. DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) approach [18] and Component Object Model (COM) [19]. DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server [20] and Active Directory [21]. DCOM provides heterogeneity across the programming language but not across operating system or tool vendor.

- **Message-Oriented Middleware (MOM):** provides the abstraction of a message queue that can be accessed across a network. It is a generalization of the well-known operating system construct. It is very flexible in how it can be configured with the topology of programs that deposit and withdraw messages from a given queue. MOM offers the same kind of spatial and temporal decoupling as in Linda model [14].

A detailed description of both CORBA and DCOM can be found in Appendix A. The next section will provide a detailed description of another widely used middleware technology called Jini, as this was chosen to support the development of the software framework described in this thesis. Jini is categorized as service-oriented middleware.

### 2.3.2 Jini Middleware Technology

From the official Jini architecture specification [22], Jini is defined as:

> *"A Jini system is a distributed system based on the idea of federating groups of users and the resources required by those users. The focus of the system is to make the network a more dynamic entity that better reflects the*

*dynamic nature of the workgroup by enabling the ability to add and delete services flexibly."*

A Jini system uses the network as a foundation to enable service discovery and execution. Traditional systems attempt to mask the appearance of the network; in contrast, Jini uses the dynamic, flexible nature of the network to form communities, register services, discover services, and invoke services. Jini attempts to address the problem of the assumptions of reliability, the network does not change, and the administrator will perform the majority of maintenance functions.

To address the assumption that the network is reliable, Jini systems are self-healing. Jini services will repair themselves by using the concept of leasing. Jini ensures that services are removed from the community automatically at a given time by using a concept called leasing, without the need for an administrator to perform the maintenance. The network can change but Jini will continue to work without the need to update a URL in the software, properties file, or an administrative interface. Jini services are able to respond to such changes automatically. Communicating with a Jini service occurs through service proxies that are downloaded to the client machine without the need to have an administrator to install the code, a practice that is normal in traditional architectures.

The Jini architecture is based on the principle that the system is dynamic, and is closely related to Service-Oriented programming [23], in contrast, other technologies such as J2EE [24], .NET [25], and CORBA have their foundation in a static world. These technologies use a static approach to installing the software with its appropriate stubs and skeletons. One of Jini's strongest points is downloadable, service-proxies that communicate with the Jini service. Jini has the same concept of a service registry, a service provider, and service requestor. The technologies previously mentioned force the SOA model to fit a static architecture that was not designed for this purpose.

The Jini architecture compromises of three main components (Fig. 2.2). There is a *Service*, such as printer, a toaster, a spellchecker, etc. There is a *Client*, which would like to make use of this service. Thirdly, there is a *Lookup Service* (service locator), which acts as a broker/trader/locator between the service and the client. There is also an additional “hidden” fourth component, which is a network connecting the previous

three together, and this network will generally be running TCP/IP (although the Jini specification is independent of network protocol).

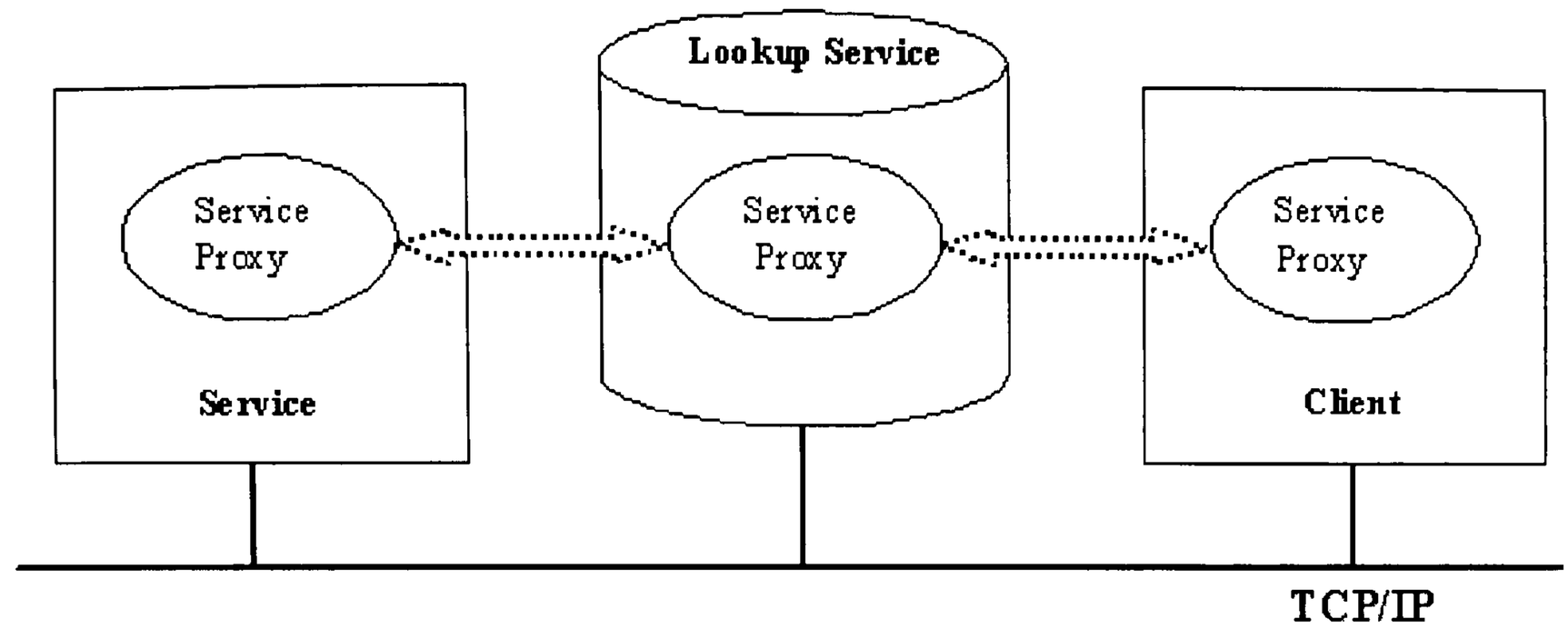



**Figure 2.3: Jini components and their interaction.**

Jini's API provides *code mobility* in that code can be moved around between these three components, over the network, by *marshalling* the objects. This marshalling involves serializing the objects (using Java's Serialization API) in such a way that they can be moved around the network, stored in a "freeze-dried" form, and later reconstituted by using included information about the class files as well as instance data. This marshalling is represented in Figure 2.3 using the block arrows with broken lines. Two events must take place in order for the client to use the application service:

- First the service, which consists of an implementation and a proxy, must register with a Jini lookup service. Sun Microsystems Jini implementation provides a lookup service (called *reggie*), which "listens" on a port for registration requests. When a request is received, a dialog between the application service and lookup service takes place after which a copy of the service proxy is moved to and stored in the lookup service.

- Second, the client must find the service. This again involves the lookup service, which also listens for incoming requests from clients that want to use services. The client makes its request using a template, which is checked against the service proxies that are currently stored on the lookup service. If a match is found, a copy of the matching service proxy is moved from the lookup service to the client.

At this stage there are three copies of the service proxy in existence (Fig. 2.2), one in the service, one in the lookup service and now one in the client. Jini service proxies

are implemented as Java interfaces that specify the signatures of methods, which are implemented by the service implementation. The client can interact with its copy of the service proxy by invoking any of the specified methods. These method invocations are then routed back to the service implementation, typically using RMI, resulting in the invocation of a method in the service implementation. The overall effect of this routing is that the client "thinks" that it has its own local copy of the service implementation and proceeds to use the service by invoking its methods being unaware of the physical location of the service within the network.

The following diagram shows (Fig. 2.4) the overall architecture of the Jini technology that is very close to Service-Oriented Architecture. The Jini architecture has a foundation that is rooted in the dynamic principles of Service-Oriented Architecture.

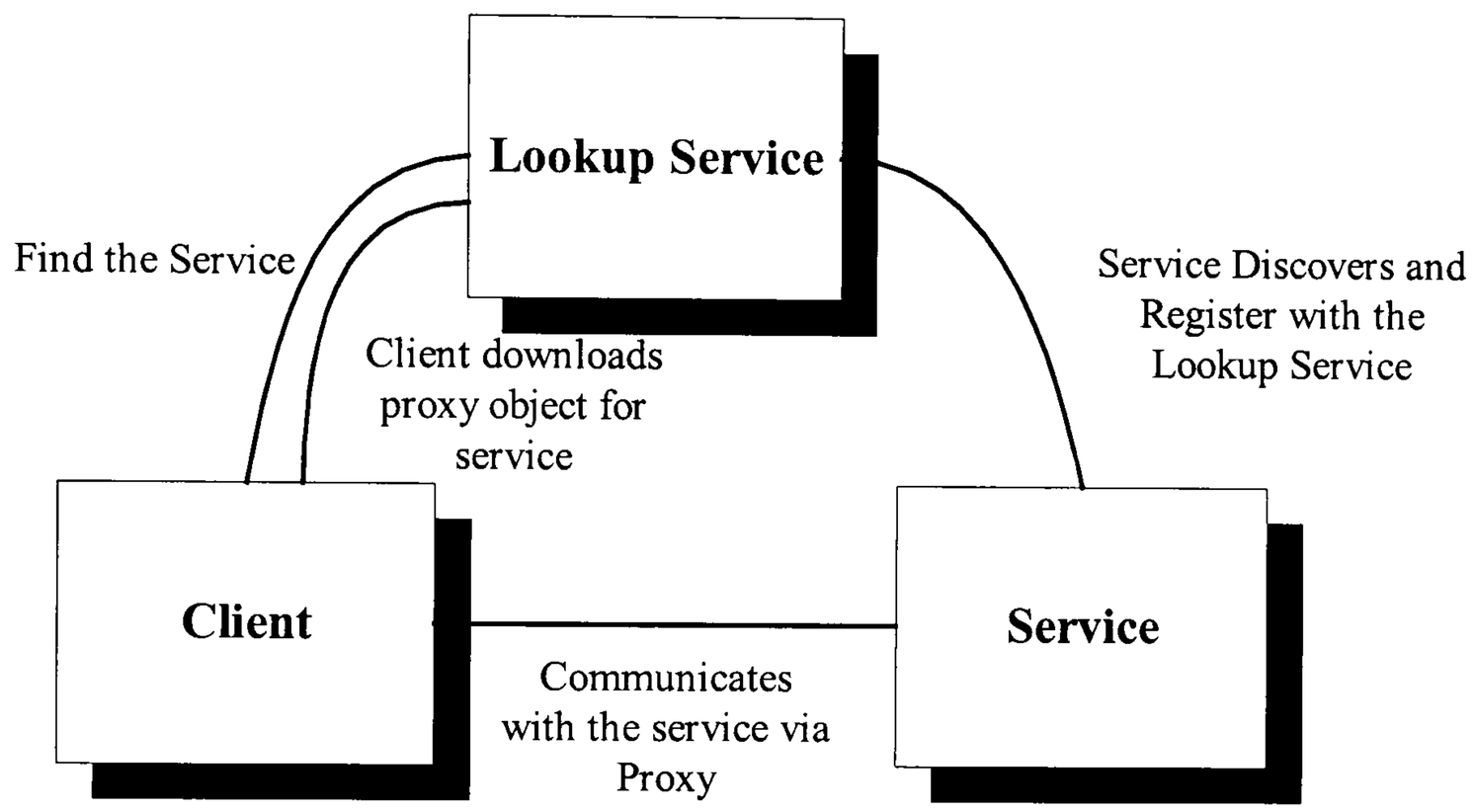



**Figure 2.4: Jini Architecture.**

## 2.4 Service-Oriented Software Development

Service-Oriented Programming (SOP), as a relatively new paradigm, builds on Object-Oriented Programming [26] by adding the abstraction that objects (components) provide and use services. In other words, where Object-Oriented Programming (OOP) focuses on what things are and how they are constructed, SOP focuses on what the things can do. A service itself is a contractually defined behaviour that can be implemented and provided by any component for use by any other component, capable of meeting the contract [3].

Component models prescribe that programming problems can be seen as independently deployable black boxes that communicate through contracts. The

traditional client-server model often lacks well-defined public contracts that are independent of the client or server implementation, which renders the client-server model "brittle". Through the service-oriented model, components may interchangeably provide and use services in a peer-to-peer manner, thereby eliminating the brittleness of the client-server model.

The new concepts of the SOP paradigm have been widely adopted and implemented within the industry, including Sun's Jini, Openwings [27] [28] and Microsoft's .NET [25]. The analyses of these technologies have yielded a set of common architectural elements that make up Service-Oriented Programming. The elements of SOP are:

- Contract – An interface that contractually defines the syntax and semantics of single behaviour.
- Component – A third-party deployable computing element that is reusable due to independence from platforms, protocols, and deployment environments.
- Connector – An encapsulation of transport-specific details for a specified contract. It is an individually deployable element.
- Container – An environment for executing components that manages availability and code security.
- Operational Space – A virtual environment containing one or more virtual containers [29].
- Context – An environment for deploying plug and play components, that prescribes the details of installation, security, discovery, and lookup.

### 2.4.1. Service-Oriented Architectures

Service-oriented architectures provide a standard programming model that allows software components, residing on any network, to be published, discovered, and invoked by each other. SOA software programmers can build services that are offered as components to anyone, anywhere via a computer network. This means that any distributed service application can interact with any other service-based application regardless of either's network location. Service-Oriented Architecture is defined as [30]:

*"SOA takes the existing software components residing on the network and allows them to be published, invoked and discovered by each other. SOA allows the software developers to model programming problems in terms of services offered by components to anyone, anywhere over the network."*

There are essentially three components and three operations to a SOA (Fig. 2.5). The components are:

- **Service Provider**: Typically the owner of the service, the service provider is responsible for publishing a description of its service to a service registry. The provider also hosts the service and controls access to it.
- **Service Requestor**: The service requestor is a software component in search of a service to invoke. The service requestor finds the service by discovering (through the Service Registry) the set of available services that meets some pre-defined criteria. Once a suitable service is discovered, the service requestor will bind to the service publisher to actually invoke the service.
- **Service Registry**: The service registry is a central repository that facilitates service discovery by service requestors. This component is optional in an SOA because it is possible for service requestors to obtain service descriptions from a variety of other sources. This includes getting it directly from the Service Provider via FTP, a URL, or some other discovery service.

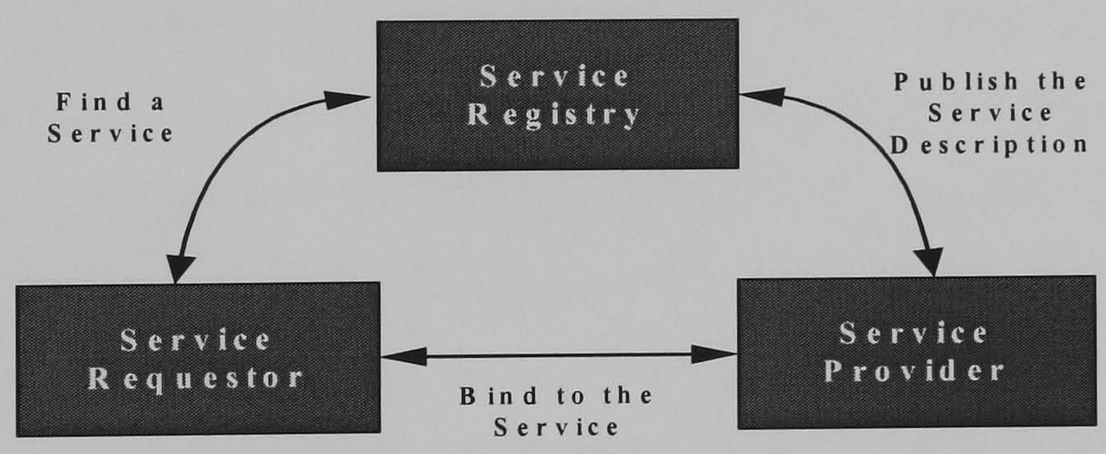



**Figure 2.5: The major components and operations of a SOA**

The major operations of a SOA are:

- **Find a Service:** An operation performed by the service requestor to locate a service. The find operation is initiated by a user through a user interface or via another service.

- **Publish a Service Description:** The service provider publishes a description of the service that is available. This description details everything necessary to interact with the service, including specific network location, transport protocols, and message formats.

- **Bind to the Service:** Once the service requestor finds an appropriate service, it will invoke the service directly at runtime using the binding information provided in the service description.

Jini middleware technology (described in the previous section) has been recognized as one of the most promising middleware technologies for implementing SOA software as Jini's architecture promotes the concept of dynamic registration, discovery, and execution. For this reason Jini was chosen as the supporting middleware for the *Impromptu framework*.

The next section explores the concept of service in relationship to the more established concept of software components, and it describes how current component-based development practices provide a tried and tested foundation for the implementation of service-oriented architecture.

## 2.5 Component-Based Development

There is growing interest in the notion of software development through the planned integration of pre-existing software components. This is often called Component-Based Development (CBD) [31], which is based on software development practices using standard components. Its origins perhaps date back to the early sixties following the publication of the "Software Crisis" report [2] and the start of the software reuse research trend [32]. The basic idea was simple: when developing new systems use components that are already developed. When we develop the specific functions that we need in our system, we should develop it in a way that allows this function to be used by other systems in the future. Although the idea and the principles are quite simple, it has been shown that the implementation is quite hard.

### 2.5.1 Components and their Interactions

The meaning of the term "component" has been changed many times during this development period. Different individuals have tried to use the word and definition

for different problem domains, relating to the technologies used at that time. The following are a selection of the definitions found in the literature today:

> *"... A component can be considered as an independent replaceable part of the application that provides a clear distinct function ..."* . [33]

According to [33], a software component is a unit of composition, with pre-defined dependencies on other components. In the world of business systems, business components represent reusable conceptual artifacts that can be implemented and deployed in large business systems. Components may also be regarded as coherent packages of software that can be independently developed and delivered as a unit. The functionality of a component is accessed via an interface and interfaces may also be used to plug components together in order to construct a larger system [34]. Components can provide services or components can use services.

For the purpose of our work, in conjunction with distributed object systems, we adopt the physical view of a component as: "*a self-contained binary implementation, which consists 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, thereby providing a *component-connector* abstraction (Fig. 2.6).

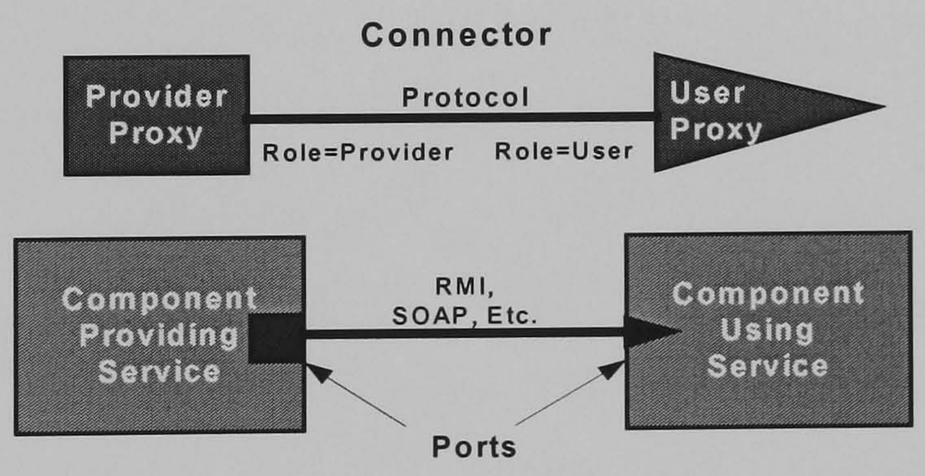



**Figure 2.6: Diagram of Component connector, etc.**

The Architecture Description Language (ADL) [35], which is a modelling language for Service-Oriented software development is based on this type of component-connector abstraction. Consequently, components may provide services and/or use services provided by other components in a peer-to-peer like regime thereby providing a federation of services, which forms the basis of the alternative *service-oriented* abstraction, which we regard as:

*"... A collection of application services spread over networked computers, which clients use remotely via distributed middleware services."* [36]

A service represents contractually defined behaviour that is provided by a specific component for use by any other component, capable of meeting the contract. The description of connectors and contract is provided in section 2.4, while the next section defines the Autonomic Computing principles and describes the architectural concepts of this type of system.

## 2.6 Extensible Markup Language (XML)

Since its introduction in late 90s, Extensible Markup Language has become ubiquitous. It is used as a base format for everything from configuration files to document management or to messages sent between computers [O'Reilly, 2006 #131]. XML combines the power and extensibility of its parent language, SGML (**S**tandard **G**eneralized **M**arkup **L**anguage) [114], with the simplicity demanded by the rapid development of Web based technologies. XML is the first language that makes the data description documents both human-readable and computer-manipulable.

The essential characteristics of XML are the data independence, the separation of content and its presentation. Because an XML document describes data, it can conceivably be processed by any application. The absence of formatting instructions makes it easy to parse. This makes XML ideal framework for data exchange. Integration of XML to any applications makes applications more dynamic and interoperating. Because XML's semantic and structural information enables it to be manipulated by any application, much of the information and the operations that were limited only to servers now it can be reachable and performable by client.

There are some alternative markup/document description languages to XML, such as: SGML, JSON (JavaScript Object Notation) [JSON, #132], SMEL (Some Modest Extensible Language) [Carlier, 2003 #139], ONX (Open Node Syntax) [Jacobs, 2005 #140], or SSYN (Structured Syntax) [Diamond, #141].

SGML was designed to be a flexible and all-encompassing coding scheme. Like XML it is a toolkit for developing specialized markup languages. But SGML is much bigger and complex than XML and it requires very complex software to process it.

Therefore its usefulness is limited to large organizations that can afford both the software and the cost of maintaining SGML environments.

JSON is a data-interchange language that is easy not only for humans to read and write, but also for machines to parse and generate. JSON is a text format that is completely language independent, but it is not as widely used, known and understood as XML.

SMEL is another XML like language that provides a method for structuring and documenting data. So is the ONX, the markup language that is designed to be data-oriented instead of document-oriented and is intended for use in platform-independent transfer of data over distributed systems, though it can be used for non-networked applications just as effectively.

The SSYN specification defines a set of both abstract and syntactic rules designed for the interchange of structured information. SSYN has been designed as an alternative to XML for the increasing number of cases where XML is currently used to store information in a rigidly structured manner. SSYN is simple and flexible language for the developers to read and write structured documents readable for both humans and machines.

Even though the number of alternatives to XML is increasing XML remains the most widely used due to its flexibility, simplicity and platform and language independence. Many leading technology developer companies such as IBM, Microsoft or Sun Microsystems use XML technology for developing frameworks that support development of services on web and message-based communication between them.

XML was chosen to support IMPROMPTU framework. The XML based service description model is one of the notions that will be described in this thesis. The service description is used for passing some necessary data from service side to client.

## 2.7 Autonomic Computing

The proliferation of Internet technologies, services and devices has made the current networked system designs and management tools incapable of designing reliable, secure networked systems and services. In fact, the development of computing systems has reached the complexity, heterogeneity, and rapid change rate. The

information infrastructure is becoming unmanageable and incapable of handling the complexity, heterogeneity and uncertainty requirements. On other hand, biological systems have developed successful strategies and techniques to handle these issues.

To solve the problems created by the complexity of existing Information Technology IBM, in 2001, started a new trend of research called "Autonomic Computing" [37]. The term Autonomic is derived from human biology. The autonomic nervous system monitors our heartbeat, checks the blood sugar level and keeps the body's temperature close to 98.6 $^{0}$F, without any conscious effort on our part. In the same way, autonomic computing components anticipate computer system needs and resolve problems – with minimal human intervention.

However, there is an important distinction between autonomic activity in the human body and autonomic responses in a computer system. Many of the decisions made by autonomic elements in the body are involuntary, whereas autonomic elements in computer systems should make decisions based on tasks the user chooses to delegate to the technology. In other words, an adaptable policy – rather than rigid hard coding – determines the types of decision and actions autonomic elements should make in computer systems.

According to IBM, any autonomic system can be characterized as follow [38]:

- To be autonomic, a system needs to "know itself" – and consist of components that also possess a system identity
- An autonomic system must configure and reconfigure itself under varying and unpredictable conditions
- An autonomic system never settles for the status quo – it always looks for ways to optimize its workings.
- An autonomic system must perform something akin to healing – it must be able to recover from routine and extraordinary events that might cause some parts to malfunction.
- As a virtual world is not less dangerous than the physical one, an autonomic computing system must be an expert in self-protection.
- An autonomic computing system knows its environment and the context surrounding its activity and act accordingly

- An autonomic system cannot exist in a hermetic environment (and must adhere to open standards).
- And a most important aspect for the user, an autonomic computing system should anticipate the optimized resources needed to meet a user's information needs while keeping its complexity hidden


**Figure 2.7: Four fundamental features of Autonomic computing [39]**

According to these characteristics, future autonomic computing systems should have one or all of the following fundamental features: self-configuring, self-optimizing, self-protecting and self-healing (Fig. 2.7).

- Self-configuring - Systems adapt automatically to dynamically changing environments. When hardware and software systems have the ability to define themselves "on-the-fly," they are self-configuring.
- Self-healing – systems discover, diagnose, and react to disruptions. For the system to be self-healing, it must be able to recover from a failed component by first detecting and isolating the failed component, taking it off line and fixing or isolating this component and reintroducing the fixed or replacement component into service without any apparent application disruption.
- Self-optimizing – Systems monitor and tune resources automatically. Self-optimization requires hardware and software systems to efficiently maximize resource utilization to meet end-user needs without human intervention.
- Self-protecting – Systems anticipate, detect, identify, and protect themselves from attacks from anywhere. Self-protecting systems must have the ability to define and manage user access to all computing resources within the enterprise, to detect against unauthorized resource access, to detect intrusions and report and prevent these activities as they occur, and to provide backup and recovery capabilities.

### 2.7.1 Architectural Concepts of Autonomic Computing

A standard set of functions and interactions govern the management of computing systems and their resources, including a client, server, database manager or Web application server. As shown in Figure 2.8, the loop represents these functions and interactions and acts as a manager of the resource through monitoring, analysis and taking action based on a set of policies.

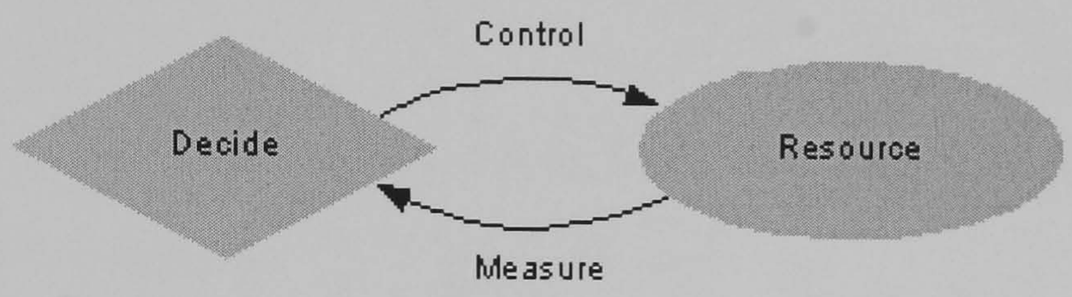



**Figure 2.8: Control loops for autonomic computing system management [39].**

These control loops 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 environment. The pyramid below represents the hierarchy in which autonomic computing technologies will operate.

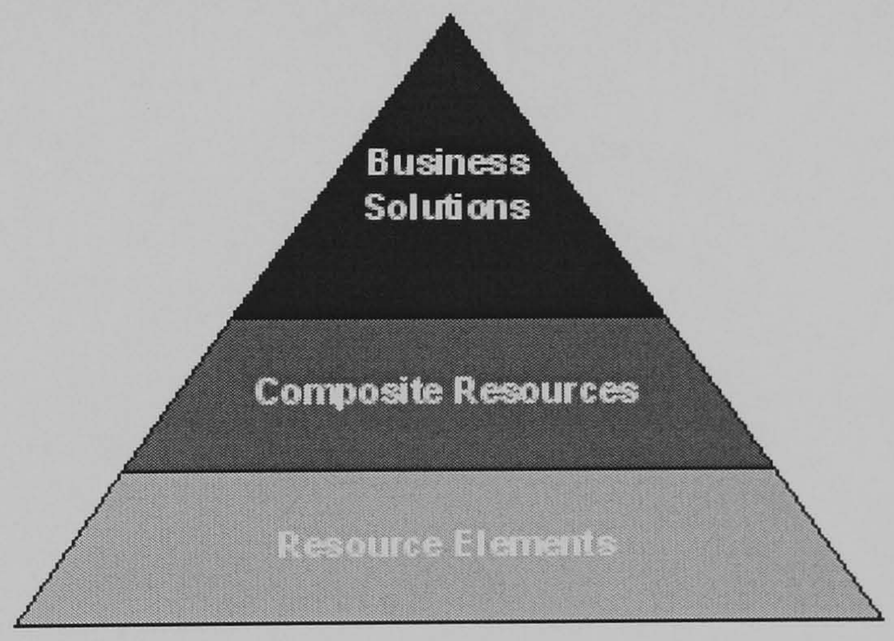



**Figure 2.9: Hierarchy of autonomic computing technologies [39]**

The *bottom 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.

The *middle layer* of the pyramid is where the resource elements are grouped into composite resources, which begin to communicate with each other to create self-

managing systems. These can be represented by a pool of servers that work together to dynamically adjust workload and configuration to meet certain performance and availability thresholds. It can also be represented by a combination of heterogeneous devices that work together to achieve performance and availability targets.

And at the *top layer* of the pyramid composite resources are tied to business solutions, such as a customer care system or electronic auction system. Autonomic activity occurs at this level. The solution layer requires autonomic solutions to comprehend the optimal state of business – based on policies, schedules, services levels and so on, and drive the consequences of process optimisation back to the composite resources and even to individual elements.

## 2.8 Summary

In this chapter we have given an overview of existing supporting technologies for distributed applications development. The basic concepts and principles of the distributed systems were described together with a brief description of middleware and common types of middleware. In addition, the Jini technology was described with reference to the services-oriented paradigm as Jini and Java used to implement the research prototypes.

# Chapter 3

# Literature Review

## 3.1 Introduction

The development of the *Impromptu* framework presented in this thesis was inspired by previous works and based on a number of paradigms, concepts and technologies existing in software development. Thus as a convenient approach to present the landscape of previous and related work the literature review is structured into three main areas including:

- Autonomic computing: focusing on the work addressing the self-healing features of systems.
- Service-oriented architectures and frameworks.
- Component-based frameworks, with a focus on those related to self-adaptive, self-healing, and self-organising software.

The Chapter concludes with an outline of the main requirements for an on-demand self-healing framework development.

## 3.2 Self-Healing Systems

Self-healing is one of the four features that characterize autonomic computing systems. Self-healing systems form an area of research that is gathering increased research attention, but, as yet, it is not well defined in terms of scope, architectural models and/or support. Still, there is a growing body of knowledge related to the general topic of dependable systems, and on techniques that can reasonably be considered to comprise "self-healing". The general view of self-healing systems is that they perform a reconfiguration step to heal a system that has suffered a

permanent fault [40]. Moreover, self-healing systems should have the ability to modify their own behaviour in response to changes in their environment, such as resource variability, changing user needs, mobility and system faults. The lifecycle of self-healing systems consists of five major elements:

- Runtime monitoring of a given target, be it the system itself or its system parts or others
- Exception Event detection including: an event arising from a deviation from a given model, normal system states and/or behaviour.
- Diagnosis including: identification of events and the right course of action.
- Generating a plan of change such as architectural transformation during a software reconfiguration process.
- Validation and enactment of a given change plan.

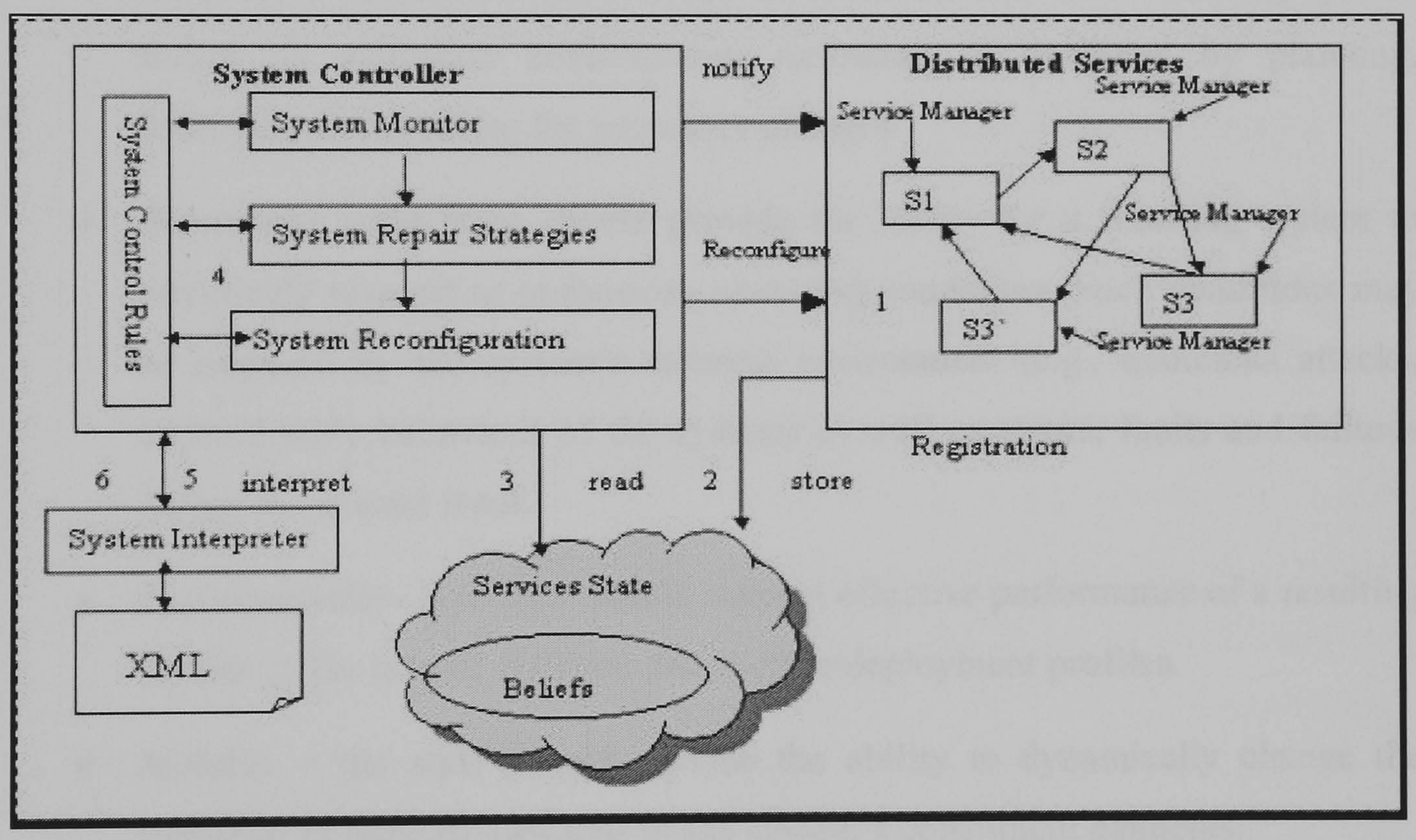



**Figure 3.1 The autonomic middleware control service architecture [8]**

The topic of self-healing systems has been studied in a many areas including: robotics-planning software, control systems, programming language design, fault-tolerant computing and software architecture. However, in this section we mainly focus on those areas related to software engineering and software architecture, highlighting how existing research relates to and has influenced the development of the Impromptu software framework.

A research group from the University of Southern California provides a well-defined architectural style and requirements for self-healing systems [41], namely:

- *Adaptability* – the style should enable the modification of the system's static (i.e., structural and topological) and dynamic (i.e., behavioural and interaction) aspects.

- *Dynamicity* - encapsulates system adaptability concerns during runtime (e.g., communication integrity and internal state consistency).

- *Awareness* – the style should support *reflection* i.e., monitoring of a system's own performance and recognition of anomalies in the performance. The style should also support *observability* i.e., monitoring of the system's execution environment.

- *Autonomy* – the style should provide the ability to address the anomalies (discovered through awareness) in the performance of a resulting system and/or its execution environment. Autonomy is achieved by planning, deploying and enacting the necessary changes.

- *Robustness* – the style should provide the ability for a resulting system to effectively respond to unforeseen operating conditions. Such conditions may be imposed by the system's external environment (e.g., malicious attacks, unpredictable behaviour of the system) as well as errors, faults and failures within the system itself.

- *Distributability* – the style should support effective performance of a resulting system in the face of different distribution/deployment profiles.

- *Mobility* – the style should provide the ability to dynamically change the (physical or logical) locations of the system's constituent elements.

- *Traceability* – the style should clearly relate a system's architectural elements to the system's execution-level modules in order to enable change enactment in support of the above requirements.

It must be mentioned that additional requirements may be relevant to certain classes of self-healing systems as they evolve; however, the above listed requirements are likely to be relevant to most self-healing systems. The author of this work hopes that the selection of specified requirements will help to identify specific, self-healing

architectural styles. However, much work remains to be done including: detailed development of the architectural styles (or models) and gaining theoretical and/or empirical experience of how to design and evaluate the applicability of such architectural styles for self-healing systems.

While the above listed requirements are increasingly accepted as essential for self-healing system development, other researchers like Gross and colleagues from Columbia University [42] offer "An Active Event Model for Systems Monitoring". This work presents a framework for communication between data-source probes and action-based gauges. It is based on an intelligent event model called *ActiveEvent (ActEvents). ActEvents* build on conventional event concepts by augmenting raw and structural data with semantic information, thereby allowing recipients to be able to dynamically understand the content of new kinds of events. Two submodels of *ActEvents* are proposed: *SmartEvents*, which are lightweight XML structured events containing references to their syntactic and semantic models, and *GaugeEvents*, which are heavier but more flexible mobile agents. By classifying the events as lightweight and sophisticated it becomes easer to deal with system monitoring.

The idea of distributed object system monitoring and supervision of a self-healing process is shared and extended in Reilly and colleagues work [43] , in which an architecture and associated middleware services were developed to support dynamic instrumentation to detect abnormal systems' states (events) and trigger and control a self-healing process thereby ensuring safety. The approach focuses on *ad-hoc* instrumentation; that is, *dynamic* software instrumentation rather than the traditional s*tatic* instrumentation. The approach used in this work makes use of instruments such as gauges, probes and monitors that can be dynamically attached to application components to measure specific runtime parameters and monitor their behaviour (Fig. 3.2). To fulfil the requirements of adaptive software development they also introduce a software control mechanism, where the control services are based on the two tasks of monitoring and diagnosis:

- The monitoring task uses a set of control rules against which monitored behaviour and architectural configuration are checked to detect conflicts.

- The diagnosis task involves the execution of control rules, activated by conflicts, which identify the causes of conflicts and provide the basis for the selection of conflict resolution and adaptation strategies.

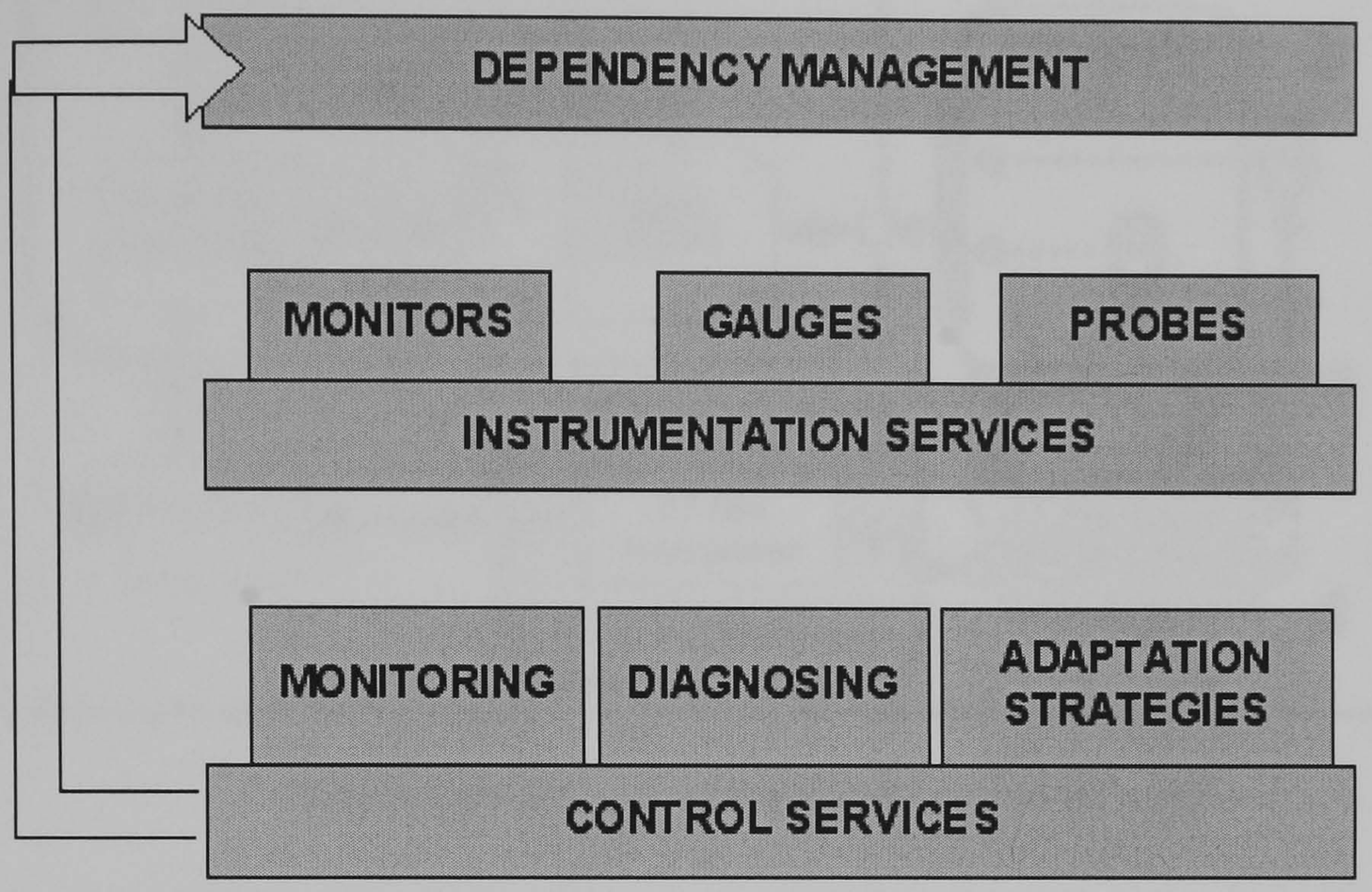



**Figure 3.2: Framework Services [43]**

The framework developed based on these services operates in a feedback controller regime in that adaptation strategies, to be performed by control services, are checked against the dependency diagraph to assess their validity. Adaptations are performed as architectural reconfigurations (i.e. diagraph modification), to the control services. Through this feedback regime the framework behaves as a self-monitoring system, the performance and behaviour of which are continually monitored to facilitate the stable operation of the application.

Garlan et al. [44] developed another architectural model for self-healing systems, based on monitoring, problem detection and repair. The use of architectural models as the centrepiece of model-based adaptation has been explored by a number of other researchers [45], but in this particular approach the architectural models are used for the runtime system's monitoring and reasoning; for instance, to understand what the running system is doing in high level terms, detect when architectural constraints are violated, and reason about repair actions at the architectural level. The approach of self-adaptation is illustrated by the following diagram (Fig. 3.3).

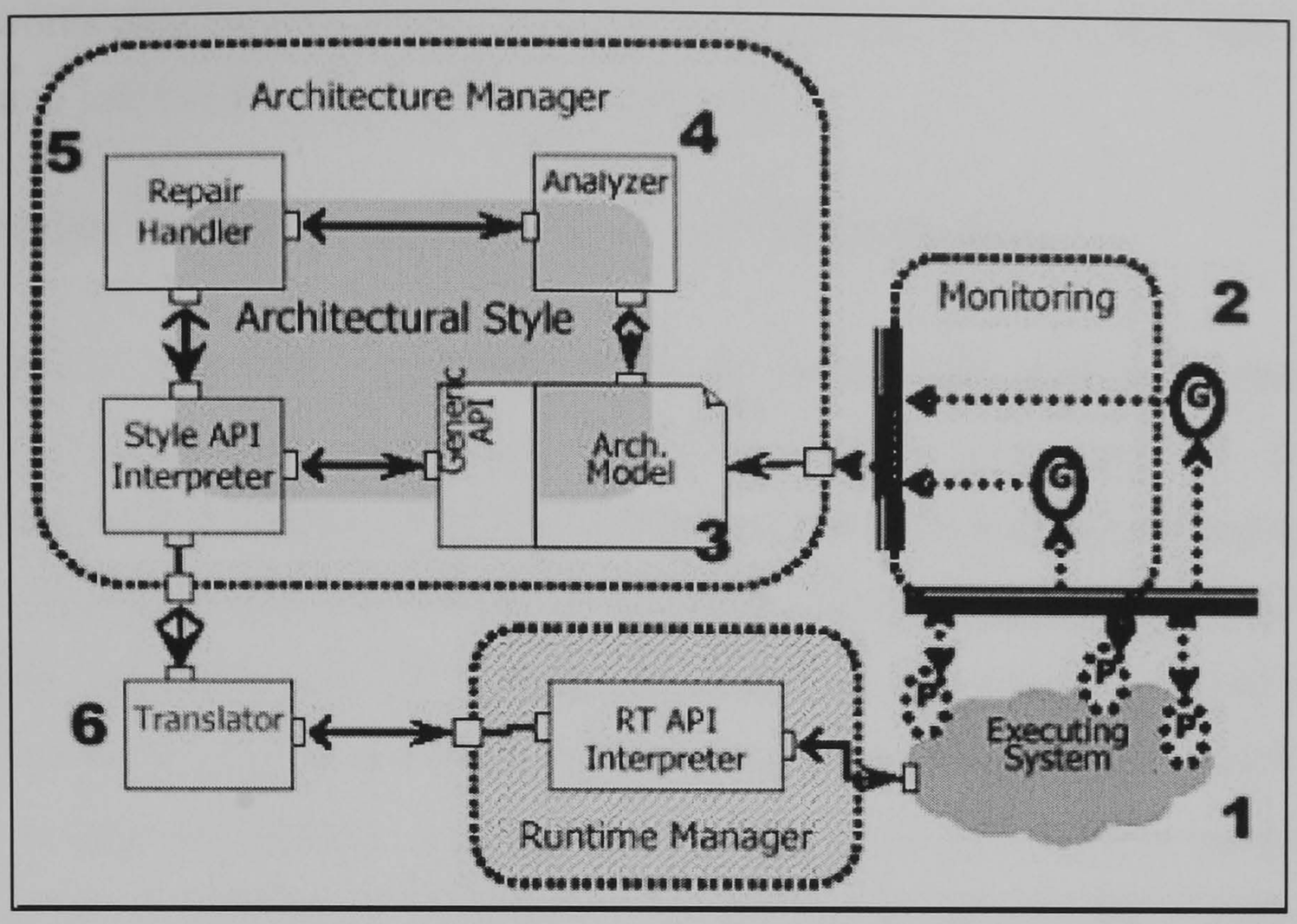



**Figure 3.3: Adaptation Framework [44].**

An executing system (1) is monitored to observe its run time behaviour (2). Monitored values are abstracted and related to architectural properties of an architectural model (3). Changing properties of the architectural model trigger constraint evaluation (4) to determine whether the system is operating within an envelope of acceptable ranges. Violations of constraint are handled by a repair mechanism (5), which adapts the architecture. Architectural changes are propagated to the running system (6).

One of the main premises of this approach is that considerable benefit can be obtained by using style-specific models within this framework. The use of style provides a focused context through which to identify conditions for repair and strategies to carry them out.

However, one style does not fit all. For different properties and different kinds of system, different styles will be relevant. This particular research focuses on distributed client-server architectures, but other applications, such as information management systems will be using different styles with associated repair policies.

There are a number of areas related to self-healing systems development such as Diagnosis and Detection, Recovery and Repair but it is not possible to cover all the works done in these areas. In the next two sections we review the work done in the areas of Service-Oriented and Distributed Component-based Architectures and

Frameworks development, respectively. In each section we review the work that is relevant to self-healing and self-adaptive software.

## 3.2 Service-Oriented Architectures and Frameworks

The Rio project [46] [47] is perhaps one of the earliest large-scale implementations of a Service-Oriented Architecture. The Rio project was supported by Sun Microsystems and based on Jini middleware and the service-oriented programming model [48]. To support Quality of Service (QoS), Dynamic Development and Fault Detection and Recovery, the Rio project introduced a number of innovations including: *Jini Service Beans (JSBs)*, *Monitor Services* and *Operational Strings*. Each of this is used to represent a collection of services and infrastructure components described in an XML-based metamodel. *Operational String* provides a ready-made specification that can be used by tools to deploy the services on the network. It also provides the capability to view, monitor and determine the availability of the aggregated collection.

For the purpose of this research, the evaluated early release of Rio software and service showed (leaving errors and bugs aside) that it supported the *static service assembly* process well: that is, the design-time assembly of application services from available networked services, and has not the support for runtime and self-healing of assembled services.

Sun ONE [49] [50] provides a comprehensive platform that includes the tools and technologies to create, assemble and deploy network-based services that are designed to support business needs – Service on Demand. The Service on Demand concept offers the creation and assembly of new services as needed. The tools used to develop systems are based on the Web services model. Web services are viewed as combinations of discrete service components that instantiate various bits of content, business logic, and applications. Therefore, the service development process involves two steps. The first step involves creating discrete services, which the ONE architecture refers to as 'micro services'. The second step involves assembling the micro services into composite services, or macro services. Developers create micro services using integrated development environments, code generators, XML editors and other tools. Service assemblers use business process modelling tools, workflow tools, scripting engines and other types of "glue" tools to assemble macro services.

Once tested and complete, the Web services are ready to be delivered to a deployment platform, and the policies and rules are ready to be deployed to an open directory that manages identity, security, and policy. However runtime monitoring and reconfiguration are not the target of this project.

The "Service and Contract language" [51] defines an architectural style for dynamically discovering and assembling services into "workflow" style service architectures. A core feature of its design is to use feedback from DASADA gauges to adapt workflow service selections. Gauges and Services are treated alike - as means of representing, adding to, and adapting workflows, on-the-fly. The Service and Contract (S + C) workflow mediates the interactions between components, DASADA gauges and tasks. Tasks are user or application requests for services; they are the stimulus for creating/using dynamic service workflows. This was used by the Cognitive Agent Architecture (Cougaar) [52], which is a Java based open source architecture for the construction of large-scale distributed agent-based applications. The role of Service and Contract workflow is to coordinate the invocation of software components and services within a dynamic multi-node environment via a blackboard architecture.

Service Providers are discovered, assembled and invoked by request. An incoming task is the stimulus by which a distributed "chain of events" is started, leading to the composition and invocation of a distributed workflow. An "S + C" workflow is a network of services (committed to achieve "a job") combined by task. Services are assembled via a "bid" process – services requested are matched to Service Providers and accepted. Acceptance depends on "Assessor". The Assessor is the part of "Service and Contract" workflow with the "Executor". The Assessor translates the output of ServiceProvider PlugIns and gives "Contracts". The Executor invokes the Contracts. When assembling services over distributed nodes, an additional infrastructure PlugIn (Service and Contract Router) is required, the Service and Contract Router loosely coordinate the workflow between nodes.

All the elements of presented architecture are the building block workflow patterns by which peer-to-peer applications can be flexibly composed. However, it can be used only with conjunction to Cougar that represents the distributed agent architecture.

## 3.3 Distributed Component-Based Frameworks

Many distributed component-based frameworks and systems already exist, each of which defines its own component model and incorporates different composition mechanism. For instance, the Openwings consortium established by Sun Microsystems and Motorola developed an architecture, standards and a framework for networked systems of software and hardware components [53]. The Openwings framework adopted the component-connector-port view of a distributed application through which components are connected together, or assembled, using protocol specific connectors that plug into ports within the components, to facilitate synchronous and asynchronous communications between components.

The Java Service Framework (JSF) [54] is an open standard, component framework that facilitates the development of servers from a set of independent but cooperating components and services. The framework defines the life cycle of these services and the ways they may locate and interact with each other. By applying Java standards in a layered infrastructure, basic components are separated and managed in a modular, controlled fashion. Using several new concepts to Java Technology, such as embeddors, a kernel and partitions, JSF supports the run-time configuration and management of services, but run-time assembly and service on-demand concepts are not the target of this framework.

The Possession System [55] is an application framework supporting component-based system reconfiguration and code migration and adopts a high-level abstraction based on the analogy that a soul possesses a body. It incorporates application-level runtime adaptation based on scripting support along with an event-notification mechanism between Soul and Body components. The Improvised Assembly Mechanism, a component of the Possession system, offers a method for finding and combining distributed resources. The basic system interaction is as follows: Body components make queries to the directory services in order to obtain identifiers of available Body components necessary to construct an application. The directory service returns available entries either synchronously or asynchronously, as well as sending asynchronous notifications when the state information on queried Body components changes. The implementation of the system is based on Jini middleware technology.

Herring describes a class of software systems as being "complex systems" [56]. Examples of complex systems include Smart Environments [57], Multi-Agent Systems [58], Adaptive/Intelligent User Interfaces [59] and Business-to-Business e-Commerce. He characterises complex systems by large numbers of heterogeneous components with a high degree of interconnection, relations and dependencies. To build this type of system he proposed a new architecture based on a cybernetic management model called the Viable System Model (VSM) [60]. This new architecture is called the Viable System Architecture [61] and defines a unique set of component interfaces that in turn defines the framework itself. It is believed that the special nature of the framework will permit development of protocols for dynamic assembly of systems from sub-system frameworks. The author of this work also proposes a new design pattern language for Viable Systems development. This work is very close to the work described in this thesis with its nature and targeted problem, but at the same time it should be noted that this work does not take under consideration the systems, or sub-systems failures and does not target recovery and reconfiguration issues. Although the variable system architecture does consider adaptation issues as complex systems existing in a dynamically changing environment and these types of environment demand dynamically responsive behaviour. Therefore adaptation in this work addresses the requirements that a system must adapt to its environment.

Self-adaptation in complex systems is widely addressed in the J-Reference model [62] [63]. The authors of this work describe the Viable System Model [60] as specifically attempting to imbue the system with the ability to adapt to circumstances not foreseen by the original designer. The model identifies the necessary and sufficient communication and control systems that must exist for any organization to remain viable in a changing environment. In doing so, the model does not attempt to specify the activities that must occur in each system. Instead, activities are typified by a cybernetic rationale to allow either the design of the activities to match the cybernetic criteria or for actual activities to be identified by their system type and hence assigned to the appropriate element of the model. The six major systems described by the VSM are as follow:

- System one (S1) – Operations

- System two (S2) – Coordination
- System three (S3) – Control
- System Three * (S3*) – Audit
- System four (S4) – Intelligence
- System five (S5) – Policy

Such a generalized approach allows the model to be applied to any organization regardless of size, and guarantees monitoring, diagnosing, re-planning and hence "perfect" management of the systems, providing both self-awareness and self-adaptive behaviours.

As was mentioned previously, multi-agent systems can be categorised as "complex" systems. A relatively simple software entity may be considered as an agent, consequently a software agent can be defined as:

> *"... software components that communicate with their peers by exchanging messages in an expressive agent communication language."* *[64]*

More typically, however, agents are larger entities that exchange information and provide services either to other agents or human users. Such interoperation allows communities of agents to address problems too extensive for a single software entity.

M. Williams and A. Taleb-Bendiab [58] used the software agent model to develop an agent-based reconfiguration framework, which provides computational constructs to facilitate functional, structural and behavioural flexibility, which enables multi-agent systems to adapt to arrange changes. The framework and its associated toolkit have been developed as a multi-agent system using the Python programming language.

In order to provide the necessary runtime adaptive behaviour and system self-awareness, two meta-languages were developed:

- Agent Definition Language - at the definition level, agents are viewed as service providers and/or consumers. Agents are therefore defined in terms of the services they provide. The services, in turn, are defined in terms of that agent. If it uses a service then it will tend to be a service provided by another agent. Thus an agent is defined by listing the services it will potentially *provide* or *use* and by listing all resources it *require*s.

- Agent Re-Configuration Language - at runtime, agents are also given the ability to re-configure themselves. This allows them to be much more adaptable to their environment.

This work can be viewed as a good foundation for not only multi-agent development but also the component-based development area. It offers all the architectural concepts of the above mentioned systems development, but the work itself was implemented in Python a few years ago and requires updating to comply with new technologies.

Olan [65] is an experimental language and runtime system, which supports connecting distributed application components. The easy construction of applications and reuse of components are mainly supported. Communication between objects is achieved using an RPC-based CS paradigm. The functionality of adaptation, however, is not targeted in the system.

In the Quality Object (QuO) framework [66], applications can be explicitly programmed with different levels of performance (i.e., different contract regions) and performance requirements by remote method invocations. It provides a QoS Description Language (QDL) and Configuration Setup Language (CSL) components to support the development of distributed applications with QoS requirements. The former associates QoS constraints to function calls and permits remote method invocations to be dispatched to alternate remote objects. The latter permits application execution to switch between different contract regions based on dynamic system conditions.

Most of the approaches described above focus on the assembly of applications or systems from distributed components and the services provided by components or agents. Others address adaptation and runtime configuration problems. However, there are many remaining questions to be addressed including;

- Is there anything new about building architectures and frameworks for service delivery, lookup, execution, assembly, or adaptation?
- What are the best approaches for delivering service on-demand and improvising applications in an ad-hoc manner?
- What if one of the components of the new assembled application fails?

Motivated by these questions, in next section we specify the requirements for the work we describe in this thesis, which we believe contributes to the development of middleware services for on-demand service delivery and self-healing systems.

## 3.4 Requirements

In line with the VSM model (Sec. 3.3) a set of design requirements are identified as necessary for the development of the proposed *Impromptu* framework for on-demand service assembly and delivery with self-healing capabilities. These requirements are detailed below and summarised in Table 3.1.

The IMPROMPTU Framework is based on Service-Oriented Paradigm (SOA) [30]. As such a number of requirements can be identified which are common to the systems that are using Service-Oriented concepts. Firstly, it is important that ***the framework complies with existing service-oriented standards (R. 1)***. This ensures a general coherency of approach to service distribution and delivery. Currently SOA is the most convenient architecture for a highly networked, dynamically changing environment.

When we talk about on-demand service delivery, it means that services should be found runtime according the user requirements. In order to do so the ***Framework shall allow services to be provided, located and used by user or software components over both local and wide area networks (R.2)***.

To locate the service Framework shall provide ability to discover and lookup services without being dependent on any specific service location (R. 3).

After service is found and located user/ or software component needs to make use of this service. One of the major requirements for this framework is to ***provide mechanism for service invocation (R. 4)***.

When service is found in wide area network it is obvious that service provider and service user are different individuals. However using any existing invocation mechanism based Jini or other middleware architectures the Client is required to know how the service was designed. In particular using any middleware technology based on Java RMI [7] the client should know the Invocation Method described in service interface. In order to do so there is need for some mechanism that will provide the client side with the additional information about service, describing the method for invocation or the location of the class file that can be reflected using Java

reflection [67]. Service signatures can be described in XML format, since XML has become ubiquitous and is used as the base format for everything from configuration files to information exchange between devices, moreover, the power of XML lies in its extensibility, and allowing developers to create their own custom XML based languages. (For more about XML see Section 2.6 and Appendix B).

For this reason in our particular case ***Framework shall provide XML based service description in order to use/ invoke the service (R. 5).***

To make a new application that is a combination of different services discovered in local or wide area network there the ***Framework shall provide mechanism for Service Assembly (R. 6)*** and it ***shall support dependency definition between the services (R. 7).*** At the same time the ***Framework shall provide a virtual space for the services to be executed and to live (R. 8).***

After the assembly takes part Framework shall supply the management mechanism with a document describing the assembly (R. 9).

In order to perform self-healing the Framework must provide the system manager that detects the service failure (R. 10).

When the failure is detected the System must respond by finding an alternative service for the failed one (R. 11) and the System must perform Reconfiguration in terms of placing the service instead a failed service (R. 12).

Table 3.1 summarise the requirements for Impromptu: Software Framework for On-Demand Service Delivery and self-healing Middleware Services.

| # | Requirements | Ref |
|---|---|---|
| **1** | The Framework shall be compliant with existing Service-Oriented standards. | **R-1** |
| **2** | The Framework shall allow services to be provided, located and used by either a user or software components over both local and wide area networks. | **R-2** |
| **3** | The Framework shall provide the ability to discover and lookup services without being dependent on any specific service location. | **R-3** |
| **4** | The Framework shall provide a mechanism for service invocation. | **R-4** |
| **5** | The Framework shall provide an XML based service description in order to use/ invoke the service. | **R-5** |
| **6** | The Framework shall provide the mechanism for service assembly. | **R-6** |

| | | |
|---|---|---|
| **7** | The Framework shall support a dependency definition between the services. | **R-7** |
| **8** | The Framework shall provide a virtual space for the services to be executed and to live. | **R-8** |
| **9** | The Framework shall supply the management mechanism with a document describing the assemblage. | **R-9** |
| **10** | The Framework must provide a system manager that detects a service failure. | **R-10** |
| **11** | The System must respond by finding an alternative service for the failed one. | **R-11** |
| **12** | The System must perform Self-Healing in terms of replacing a failed service | **R-12** |

**Table 3.1: Requirements for Impromptu software framework for on-demand service assembly and delivery and self-healing Middleware Services.**

## 3.5 Summary

This chapter has presented an overview of relevant research areas to this work. These can be categorised in terms of three research areas: (i) Autonomic Computing focusing on the self-healing features of systems, (ii) the service-oriented architectures and (iii) frameworks and distributed component-based frameworks. We discussed how different researchers or commercial bodies addressed similar problems and what techniques and approaches they use to solve existing problems.

Considerable research has been undertaken in the area of distributed systems development, using service-oriented concepts and architectures [68]. These model software components as service providers or service users and divide the software into small reusable components and services and provide the means to construct systems using those distributed components.

We also described some closely related work on self-healing and self-awareness. Most of this work is concentrated with how to provide software with self-adaptation, self-control and self-management.

Table 3.2, outlines the results of a comparative review of the *Impromptu* framework against published related frameworks (references are provided below). For the illustrative purposes 8 most related works were selected for the table, the rest of the related work is reviewed in main text of this chapter. The review was conducted along the general requirements defined for *Impromptu* software framework.

| # | Requirements for Impromptu Framework | Ref | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | The Framework shall be compliant with existing Service-Oriented standards. | R-1 | √ | √ | √ | √ | √ | √ | √ | √ |
| 2 | The Framework shall allow services to be provided, located and used by either a user or software components over both local and wide area networks. | R-2 | --- | --- | --- | --- | --- | √ | √ | √ |
| 3 | The Framework shall provide the ability to discover and lookup services without being dependent on any specific service location. | R-3 | --- | --- | --- | --- | --- | √ | √ | √ |
| 4 | The Framework shall provide a mechanism for service invocation. | R-4 | --- | --- | --- | --- | --- | √ | --- | √ |
| 5 | The Framework shall provide an XML based service description in order to use/ invoke the service. | R-5 | √ | --- | --- | --- | --- | --- | --- | --- |
| 6 | The Framework shall provide the mechanism for service assembly. | R-6 | --- | --- | --- | --- | √ | √ | √ | √ |
| 7 | The Framework shall support a dependency definition between the services. | R-7 | --- | √ | --- | --- | √ | --- | --- | --- |
| 8 | The Framework shall provide a virtual space for the services to be executed and to live. | R-8 | --- | --- | --- | --- | √ | --- | --- | --- |
| 9 | The Framework shall supply the management mechanism with a document describing the assemblage. | R-9 | --- | --- | --- | --- | √ | --- | --- | --- |
| 10 | The Framework must provide a system manager that detects a service failure. | R-10 | √ | √ | √ | √ | --- | --- | --- | √ |
| 11 | The System must respond by finding an alternative service for the failed one. | R-11 | --- | --- | √ | --- | --- | --- | --- | √ |
| 12 | The System must perform self-healing in terms of replacing a failed service | R-12 | --- | --- | √ | √ | --- | --- | --- | --- |

**Table 3.2 Assessment of related work against Impromptu framework**

**Key:**

1 - An Active-Event Model for System Monitoring [42].

2 - An Instrumentation and Control-Based Approach for Distributed Application Management and Adaptation [43].

3 - An Investigation into Autonomic Middleware Control Services to Support Distributed Self-Adaptive Software [9];

4 - Model-Based Adaptation for Self-healing Systems [15].

5 - Rio Project [69].

6 - Sun ONE [49];

7 - The Service and Contract (S + C) [70].

8 - The Possession Systems [13].

As it is illustrated by, Table 2.3, few frameworks have considered runtime service discovery and invocation requirements to support self-healing and self-adaptation capabilities.

The following chapters will describe how the above listed requirements were met, implemented and tested.

# Chapter 4

# Impromptu: Software Framework for On-Demand Self-Healing Middleware Services

## 4.1 Introduction

This chapter presents the overall design of *Impromptu*: Software Framework for On-Demand self-Healing Middleware Services. The framework supports both on-demand distributed service discovery and assembly, and self-healing in terms of discovery and replacement of a failed component. This chapter starts with an introduction of the fundamentals, concepts and approaches used in the process of design and development of the system. This will be followed by a description of the software development model assumed by the research and the main component services of the framework.

## 4.2 The Overall Design of the Framework

In recent years, much attention has been focused on design approaches, processes, and tools to facilitate the development of large software systems through the assembly of Commercial Off The Shelf (COTS) software. If we consider the fact that operating systems are distributed across many machines to improve performance, availability and scalability, such a system can be considered to be composed of distributed services that offer different functionality. Exposing functionalities as services is the key to flexibility. This allows other pieces of functionality, implemented as services, to make use of other services in a natural way regardless of their physical location. A system evolves through the addition of new services. In order to design and develop such systems, specific architectures and approaches are needed. This thesis explores software services in relation to the concept of software

components, and describes how current component-based development practices provide a tiered foundation for the implementation of a service-oriented architecture. In simple terms, a service-oriented architecture provides a standard programming model that allows software components, residing on any network, to be published, discovered, and invoked by each other. A Service-oriented approach, in conjunction with component-based development, provides the basic building blocks for the *Impromptu* framework. The next section describes, in more detail, the framework approach with components and patterns.

### 4.2.1 Components, Frameworks and Patterns

The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present day applications [71]. One reason is that, although object-oriented methods encourage one to develop rich models that reflect the objects of the problem domain, this does not necessarily yield software architectures that can be easily adapted to changing requirements. In particular, object-oriented methods do not typically lead to a design that makes a clear separation between computational and compositional aspects [72].

As early as 1969, McIlroy [73] proposed an alternative to software development called component-oriented software construction. However, in the last few years, component-based software development has become very popular. Component-based systems achieve flexibility by clearly separating the stable parts of the system (i.e. components) from the specification of their composition. Components are black box entities that encapsulate services behind well-defined interfaces. These interfaces tend to be very restricted in nature, reflecting a particular model of plug-compatibility supported by a component-framework, rather than reflecting the rich, real-word complexity of the application domain. Components are not used in isolation, but according to a software architecture that defines the interfaces that components may have and the rules governing their composition. Also, the architecture and rules are defined by the framework. These two terms: *Component* and *Framework* are closely related as Markus Lumpe *et al.* define [74]

> *"A software* component *is a composable element of a component* framework"

Although this definition seems to be circular, it captures the essential properties of components: components are designed to be plugged together with other components. A single component that does not belong to a component framework is a contradiction in terms. Furthermore, a component can, in general, not function outside a well-defined framework. Therefore, a component framework can be defined as follows:

> *"A* component framework *is a collection of software components with a software architecture that determines the interfaces that components may have and rules governing their composition". [75]*

This definition closes the loop. The essential point is that components are not used in isolation, but in accordance with a software architecture that determines how components are plugged together. In contrast to an object-oriented framework, where an application is generally built by subclassing framework classes that respond to specific application requirements (also known as *hot spots* [76]), a component framework primarily focuses on object and class composition (i.e. *blackbox* reuse).

Another definition of the term component framework is given by Szyperski [33]. He describes a component framework as a set of interfaces and rules of interactions that govern how components plugged into the framework may interact. He also points out that an overgeneralization of that scheme has to be avoided in order to keep actual use of frameworks practicable.

Naturally, it is not enough to have components and frameworks for building real applications, there is also a need for a mechanism to wire components together (i.e. to express *compositions*). The idea behind component-based development is that an application developer only has to write a small amount of *wiring code* in order to establish connections between components. It can also be built by defining the patterns that describe the structure of the framework. Such patterns can guide the application developer to the solution of a particular problem.

Patterns have recently become a popular way to reuse design information in the object-oriented community [77]. A pattern is an essay that describes a problem to be solved, a solution, and the context in which that solution works. Patterns are supposed to describe recurring solutions that have stood the test of time.

Since some frameworks have been implemented several times, they represent a kind of pattern, too. Gamma describes patterns as very closely related to frameworks [77]. Such patterns were discovered by examining a number of frameworks, and were chosen as being representative of reusable object-oriented software. In general, a single framework will contain many of the patterns, so these patterns are smaller than a framework. On the other hand, patterns are more abstract than frameworks. They describe solutions without being tied to one specific programming language, whereas frameworks are implemented in one and only one programming language. Still design patterns represent the architectural elements of the framework.

The framework described in this thesis is implemented in one specific language although the concepts are generic and it can be implemented using different languages and techniques, as long as this language or technique provides necessary tools to solve this specific problem. For this purpose, we describe the kind of patterns that should be used to develop a system for "on-demand service delivery" assembly with self-healing behaviour.

The next sections will describe the model of an on–demand service assembly and delivery mechanism with self-healing behaviour. Further description of the pattern language will be provided in Chapter 7.

## 4.3 Software Development Model

The software development model assumed by this research is based on a service-based development paradigm [23], also referred to as the service model [116] in which services are viewed as an abstraction or virtualisation of software components. In such a model, software services are provided by software components, and interact with each other to provide or use (consume) data and behaviour. One of the benefits of service-oriented approach to software development is that at any given time, a wide variety of alternative services may be available that meet the needs of a user.

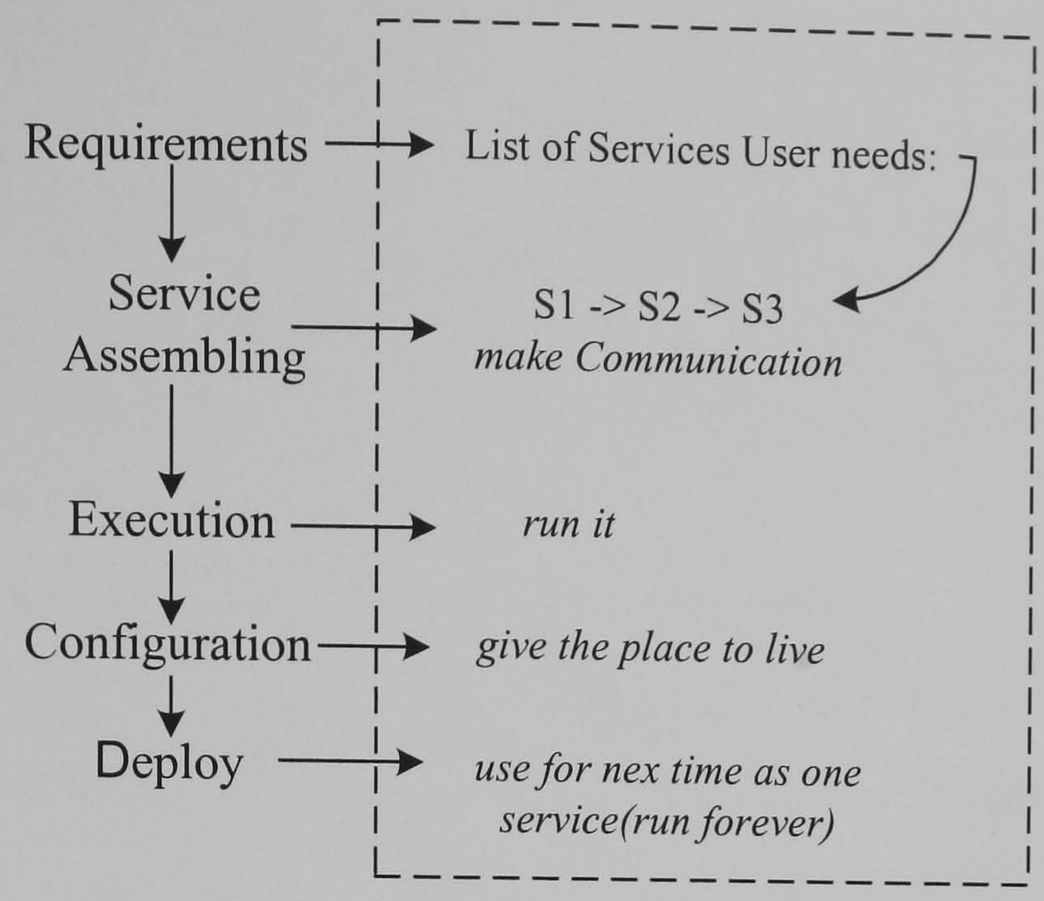



**Figure 4.1: Self-Assembly Process**

As illustrated by Figure 4.1, the design process can be seen to start from level one where the user requirements are defined in this approach as a list of required services. These services will be discovered and activated (via dynamic service invocation).

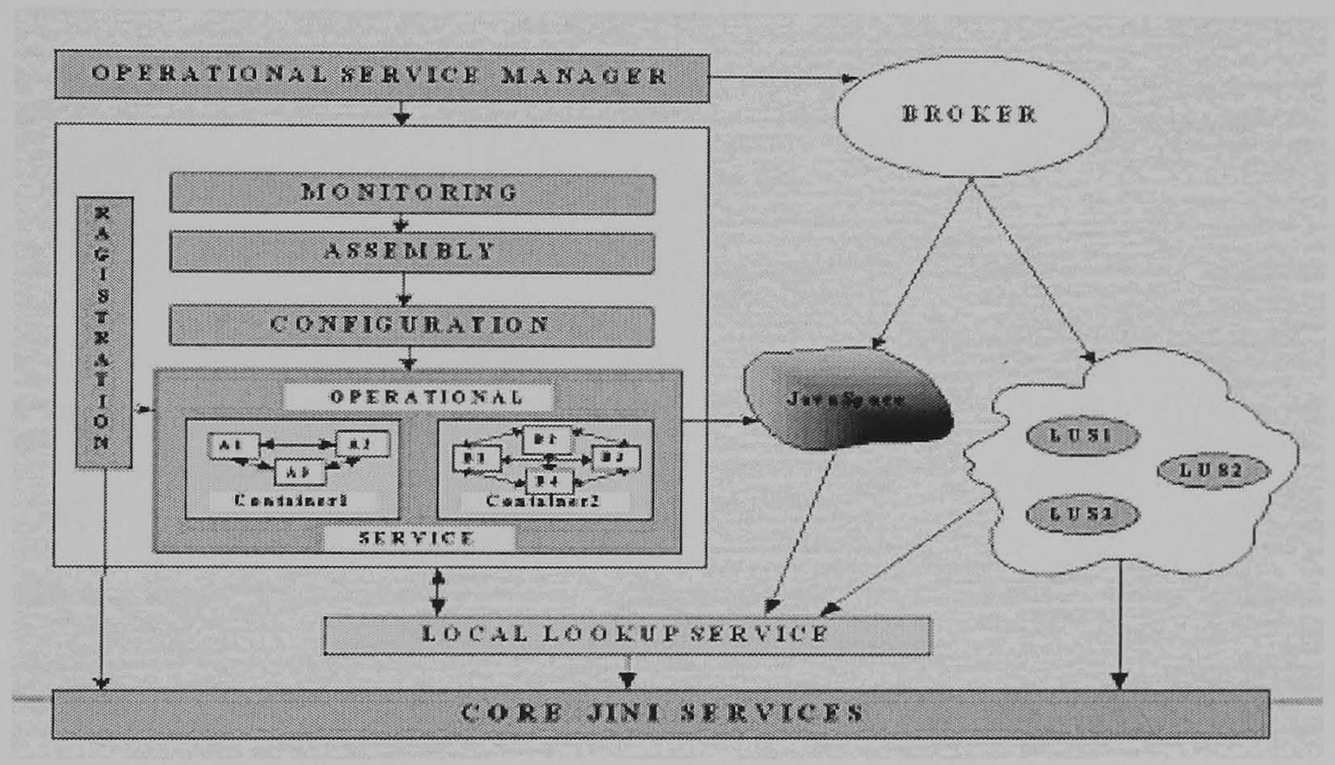



**Figure 4.2: Application Services and tools**

For example Figure 4.2 shows what services and tools are required for an application to be composed of distributed services. A discovery mechanism is needed to find distributed services that match to user's requirements. Distributed services that compose a distributed application may fail unpredictably, for this reason other mechanisms like self-healing including failure detection and recovery are needed in line to discovery properties. Using all these properties software will be able to self-heal (complete the given task) by detecting the failure of the services and rediscovery of the component that was providing the service, or other components that provide similar services.

Runtime service delivery, integration and management properties to some extent are achievable by using existing middleware technologies. Jini is one of the technologies that provide the necessary support for developing service-based distributed applications. The software development model assumed by this research enables the incorporation of self-healing properties in distributed service-based applications through the extension of middleware technology.

Jini middleware technology provides a number of distributed system services, including discovery, lookup, remote event management and service registration. When a service is plugged into a Jini network, it becomes registered as a member (service) of the network by the Jini lookup service. When a service is registered, a proxy is stored by the lookup service. The proxy can later be transported to the clients of the service. Other network members can discover the availability of the service via the lookup service. When a client application finds an appropriate device, the lookup service sets up the connection. In our approach to component integration, we use Jini to provide a standard method for registering and connecting a client to corresponding software components that are acting as services. One of the advantages of using this Jini-based integration technique is that it facilitates construction of applications ''on-the-fly'' whereby components can be used on an as-needed basis. One of the disadvantages is that clients of services must have some prior knowledge about how to use each respective service. To overcome this disadvantage of Jini, using our software model services are registered with the additional information. In other words service providers register services with a document describing all the necessary information (service signatures). Hence, a service user has a prior knowledge about how to use a particular service.

In service-based distributed systems, new services may be added or removed from the network, this leads to the need for applications to reconfigure themselves. Therefore, it is necessary to incorporate self-healing properties in applications, which are built by integrating distributed services. Service-based distributed applications developed with the support of the framework described in this work will have self-healing properties, as the framework provides service monitoring and recovery services that act as middleware service in extension of existing middleware services. The following figure shows the IMPROMPTU framework services as extension of Jini middleware technology in relationship with distributed services that are provided and used by distributed software components.

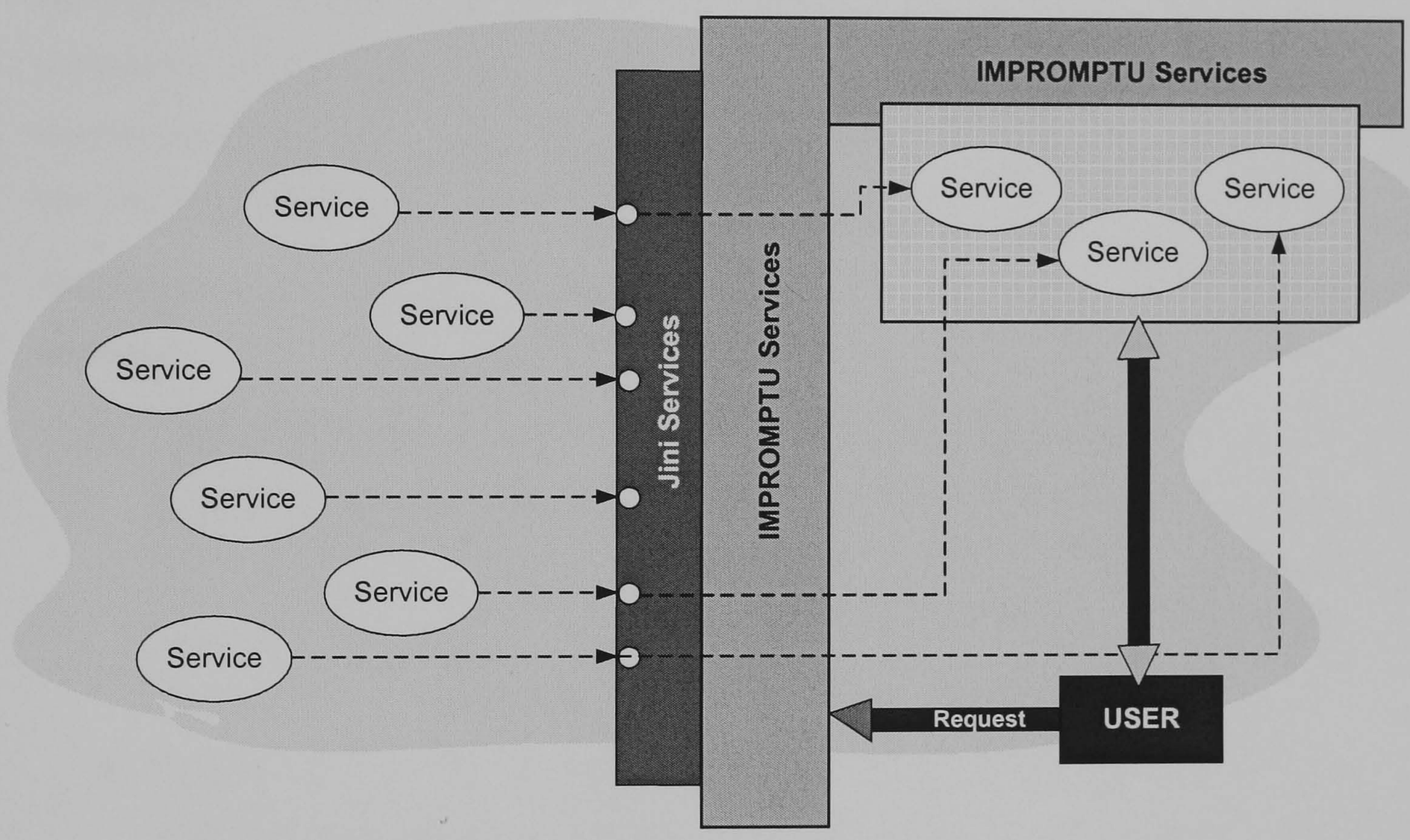



**Figure 4.3: High-level view of application development using distributed services**

The Impromptu framework consists of a collection of services that can be added to Jini services in order to enhance service registration and discovery, assemble applications by incorporating the discovered services and add self-healing properties through monitoring and failure detection of assembled services. As such, Impromptu middleware supports the software development model described in this section.

The development of an IMPROMPTU application can be described as follows: a user registers service into a lookup service, which can be achieved in two different ways: (i) if the location of the lookup service is known, then service provider can use

unicast TCP to connect directly to it. (ii) If the location is unknown, the service provider will make UDP multicast request, and lookup service may respond to these request. Service needs to be implemented with pure Java code, but for registration it is necessary to create Java Interface to invoke method on an object. To make easier for user to convert pure Java code to Jini code toolset provides some kind of template that helps user without any knowledge about Jini use the toolset.

In order to manage services it is necessary to monitor them. Services will start and stop. When they start they will inform the lookup services, and sometime after they stop they will be removed from the lookup services. But very often other services or clients want to know when services start or are removed. Monitor service that part of the framework is the service itself and gets registered on the lookup service. The availability of the service is known for the System manager/user through the monitoring service. The system manager is also responsible for recovery of the failed services by replacing it with the closet alternative service.

The sequences of the steps for a user's job could be classified as three different scenarios:

- Case I: The user has a set of Java-based services and wants to use the Impromptu framework to assemble new services from them. Using the Java-based templates provided with the framework the user can modify the Java code, and then register the newly assembled service with its local lookup service, this way the new services will also become available for other users.
- Case II: The user has no available services, in such case he looks for them on different lookup services. When found the service assembly as in Case I continue takes place.
- Case III: The user modifies existing code using the template but for completion of a new application he needs the other services. He finds it on any lookup service that is available in network at that time. Thus combining case I and II.

In the remainder of this chapter, we describe the various components of the Impromptu framework.

## 4.4 On-Demand Service Assembly and Delivery Model

Service on-demand is derived from the recognition that many different types of services will be delivered over electronic networks, including a new breed of applications that extend well beyond the tightly coupled services available today. The service on demand concept is the foundation for modular, flexible, automated access to any digital device, including computing resources, from virtually anywhere.

The On-demand Service Assembly and Delivery (OSAD) model provides an abstract view of the relationship of distributed components and services. The objective of the OSAD model is to organize the following issues in a uniform framework;

- on-demand service delivery and invocation regardless of the location of the service,
- the automatic assembly of the application in an ad-hoc manner based on the user's requirements,
- the ability of self-healing at runtime in terms of replacing the failed component of an Impromptu application.

The Impromptu application is basically a composition of different services invoked from different locations[3] and interacting with each other using remote event notification. These applications can be formed in local or wide area network, depending on the tools and technologies that will be used for implementation of *Impromptu* model. In this work Jini middleware was chosen to support the development of *Impromptu* model, therefore network coverage for *Impromptu* applications should be used for Wide Area Network services [78].

As previously mentioned all services are provided or used by distributed software and hardware components. One of the tasks of the model is to find distributed components offering specific functionality (called offering the service). After the component is found the next task is to use this functionality.

---

[3] Mainly designed and tested for Local Area Network services. Though, it can be used for Wide Area Network services.

To describe these events, we use the term 'on-demand service delivery'. The OSAD model can be described as a combination of different building blocks - component services. Figure 4.4 shows the generic model, including all of the building blocks and functionalities.

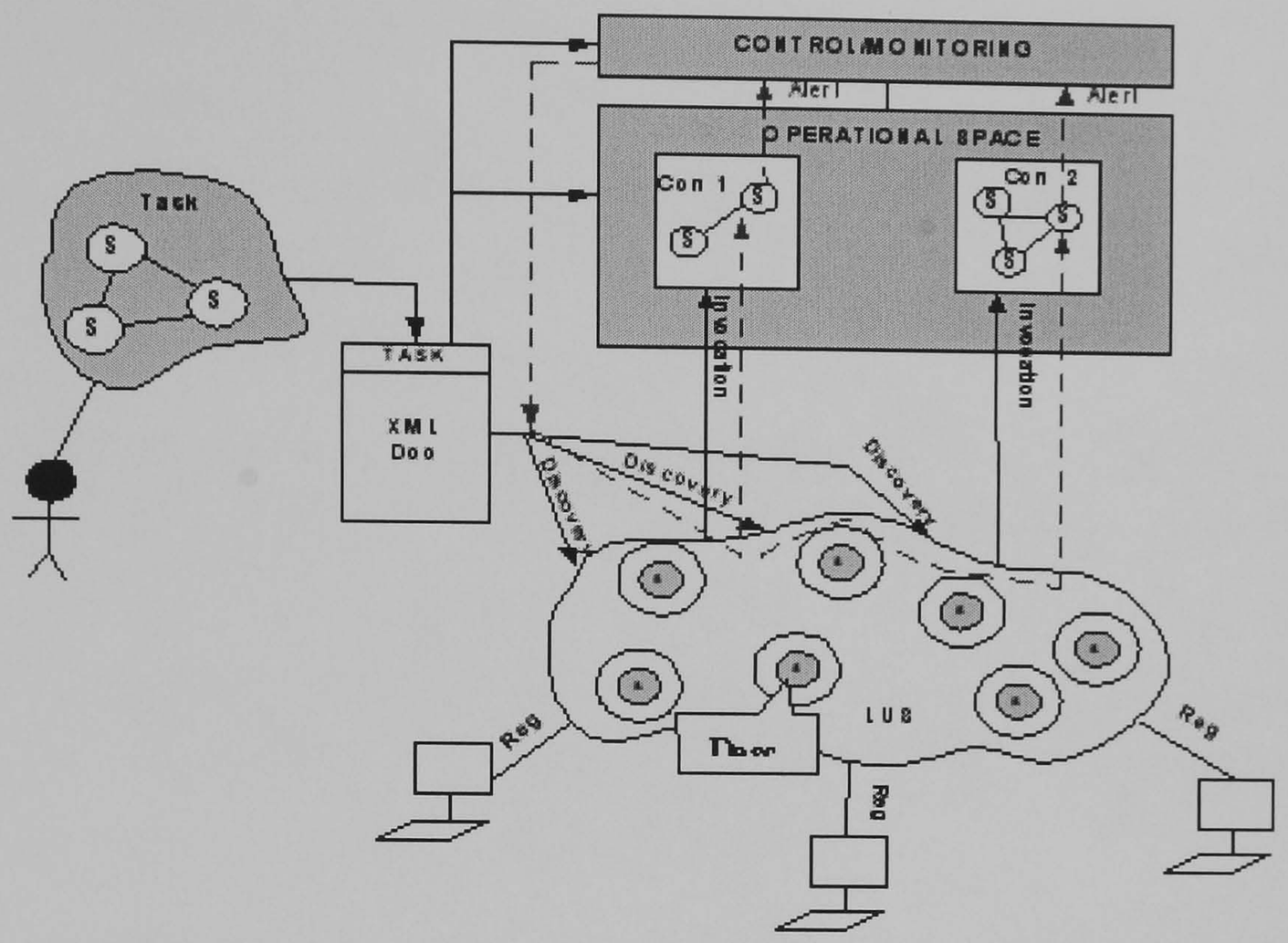



**Figure 4.4: OSAD Model**

The starting point for the model is the user defining the requirements as a task. A user enters the names of desired services and defines the dependencies between them. This will be documented and saved for further use.

The Lookup service is one of the major components of the model. This is where distributed components register the services they provide. On the other hand, the same services should be discoverable from the client side. After discovery, service invocation should take place. Invoked services require a place to "live". The Operational Space is a virtual space with Containers and the Containers are virtual places where the services live during their life cycle.

Control and monitoring is needed to identify failure and alert the system to find a replacement for the failed service, so the control mechanism is implemented with self-healing behaviour.

Based on this model, three main services were designed for the further development of the framework. These services are:

- **Assembly Service** – this is the core service of the framework and it combines a number of functionalities of the model. The Assembly service is combination of different sub-services and modules. It contains:

  *A Task Definition service* – that enables the user to define and submit a task. In this work we define a task as the requirements of the user. In other words, the user makes a request for the services s/he requires. With the user requirements in place, the system will begin to function. This means that the system will start looking for services to invoke and provide the service functionality to the user.

  *Registration and Discovery Services* - although these services are not new concepts and are provided by a number of existing technologies, it must be mentioned that these services are major building blocks/components of the model. Without this service in place, no distributed service can be registered on the network, therefore no services could be discovered and used.

  *Service Invocation Service* – is responsible for invoking any discovered service regardless of the provider and location. To implement this service we have proposed and developed a new service description mode, which is described further in Section 4.4.2.

- **Operational Service** – is the virtual space containing a number of containers. The services must be executed and assembled in these virtual containers. Having virtual containers in this type of framework enables the system and user to monitor and manage the new Impromptu applications.

- **The System Manager** – the concept of a system manager is very widely used and therefore dose not respond to one definition. In this work, the system manager represents a combination of two services:

  *Monitoring service* - is responsible for monitoring, in other words, look after the new applications.

  *Recovery/Reconfiguration service* – is responsible for recovering the application from failure. This means finding an alternative service to replace the failed service, that is, the system will search for an

equivalent service – hot swapping. The replacement process involves reconfiguration functions.

Adaptation is provided to the system by self-healing behaviour and is described further in Section 4.5. The next section defines the concept of the Impromptu service and includes the basic service of the OSAD model.

### 4.4.1 Basic Services in the OSAD Model

The OSAD model concentrates on two types of services: Software services provided by distributed software components and hardware services provided by network appliances.

> *"A service in general represents physical and logical concepts such as a printer (hardware service) and a Calculator, Text Editor (software service)"*

For effective use of services, two main characteristics of the OSAD model services are:

- **Interface based design** – Services implement separately defined interfaces. The benefit of this is that multiple services can implement a common interface and a service can implement multiple interfaces.
- **Discoverable** – Services need to be found at both design and run time, by unique identity, interface identity and also by services kind.

In both component and service development, the design of the interfaces is performed so that a software entity implements and exposes a key part of its definition. Therefore, the concept of "interface" is the key to a successful design in both component based and service-oriented systems [68].

The interface defines a set of public method signatures, logically grouped but providing no implementation. An interface is a contract between the requestor and provider of the service. Moreover, the Published Interface is uniquely identifiable and available through the registry.

Using existing middleware technologies, it is already possible for service providers to publish interfaces on some shared space in a local or global network, and for the client to discover these interfaces. But getting the set of method signatures from the

interfaces remains the problem. The next section describes the notion of a service description model.

### 4.4.2 Service Description Model

Recently, many different service description languages and models have been proposed. Web services offer an XML-based service description language, describing networked services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information [79]. The Rio project tends to use the same XML format to describe the aggregation of services rather than the individual service [80]. All of these were described in previous chapters. This section revisits the XML based service description mode that was developed for the OSAD model. Our Service description model was motivated by a problem that has not been addressed in previous works.

As previously described, to make use of a distributed service, discovery of the services is not enough. When we talk about distributed services, it is obvious that the service provider and client of the same service are different individuals. So it is impossible for the client to know the method signatures defined in the interface of the service. Without knowing this, it is also impossible to implement the interface. To solve this problem, we offer a very simple model for service description, formatted in XML, that describes the method signatures with the name and interface name of the specific services. This information will be saved on the server and the location can be passed to the client as an attribute of the service registered in the lookup space. The client can discover the service using this additional attribute.

The basic format of the service description is illustrated in the Figure 4.5.

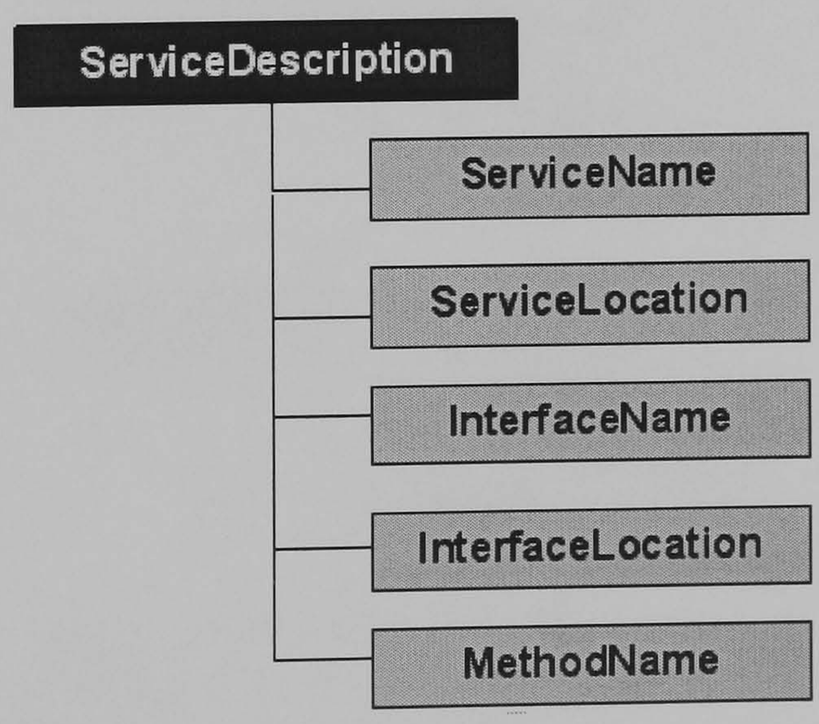



**Figure 4.5: Service Description Model**

As shown in Figure 4.5, every service should be described using the following attributes:

- ***Service Name***, a required attribute, providing a human and computer readable identifier for each service.
- ***Service Location,*** the actual physical location of the service.
- ***Interface Name,*** a required attribute, providing a human and computer readable identifier for the service interface. The interface where service signatures are described.
- ***Interface Location,*** the actual physical location of the service interface file.
- ***Method Name,*** a required attribute, providing a human and computer readable identifier for service execution.

This is the basic model for service description that is needed to pass invocation method names from the provider to the executor (human or computer).

A similar type of description is used to describe and document the task the user defines (Fig. 4.6).

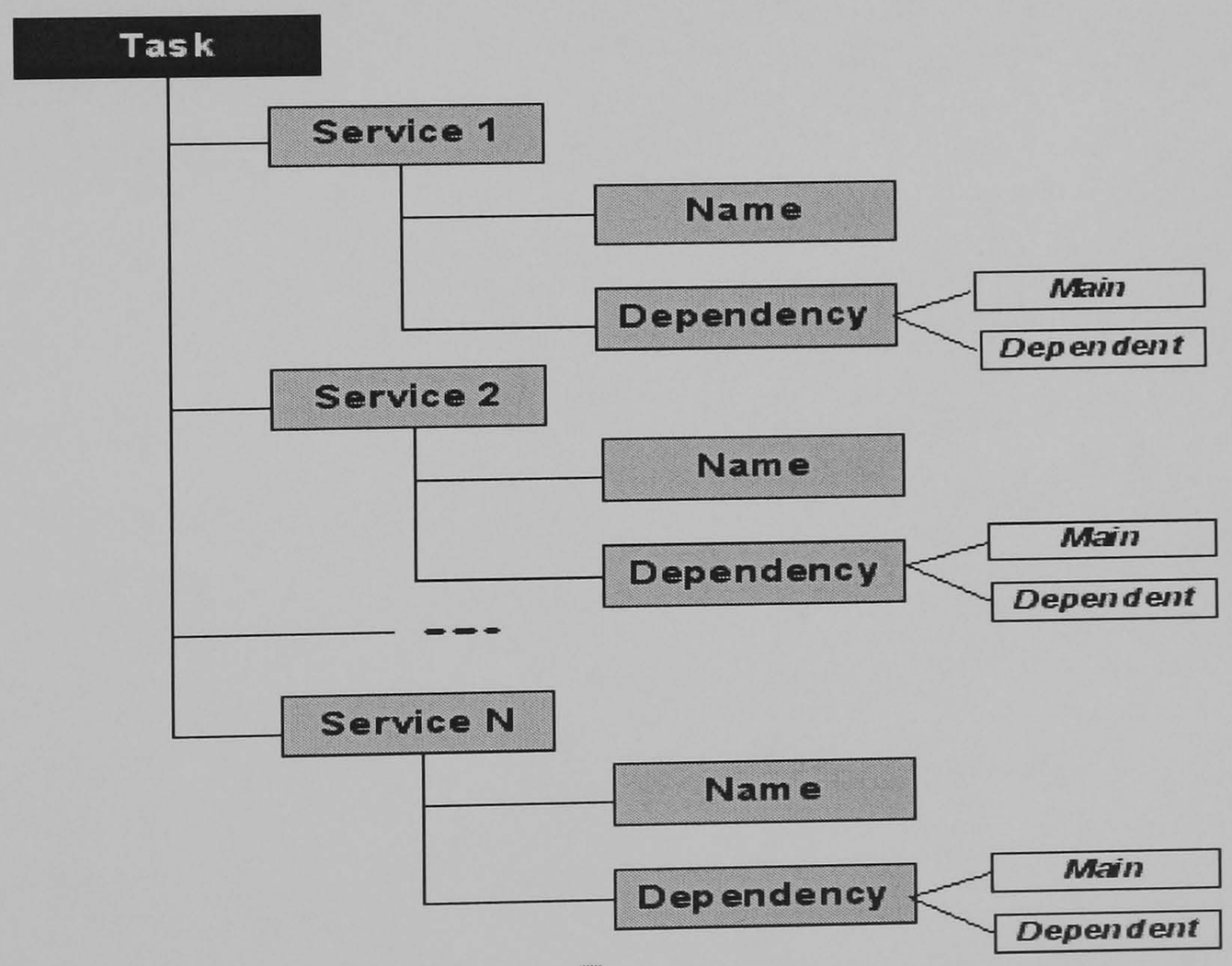



**Figure 4.6: Task Description Model**

The main attributes the user needs to define in order to submit his/her requirements are:

- ***Service Name,*** the attribute that is required for the system to make discovery and service matching.
- ***Dependency,*** the user is required to differentiate between the main service and the dependent service.

A task described and submitted in this format is readable by the system. According to the information specified in the task description, the system manager is able to identify the main service from dependent service. Making a decision about finding the new service to replace a failed one will be dependent on this information. If the main service fails, then the alternative service should be found immediately.

The distributed application can be assembled according to the task description provided by the user, so the system does not require further input from the user. Alternatively, the user can interact with the system in order to find the desired services, invoke them and place then in specific containers. Although these steps could be done manually, in terms of application monitoring and control, the users' interaction is no longer required. However, the user should submit some kind of description of the current state of the application. From this description, the system will know what services were executed and in which location. The format of this application description model is shown in Figure 4.7.

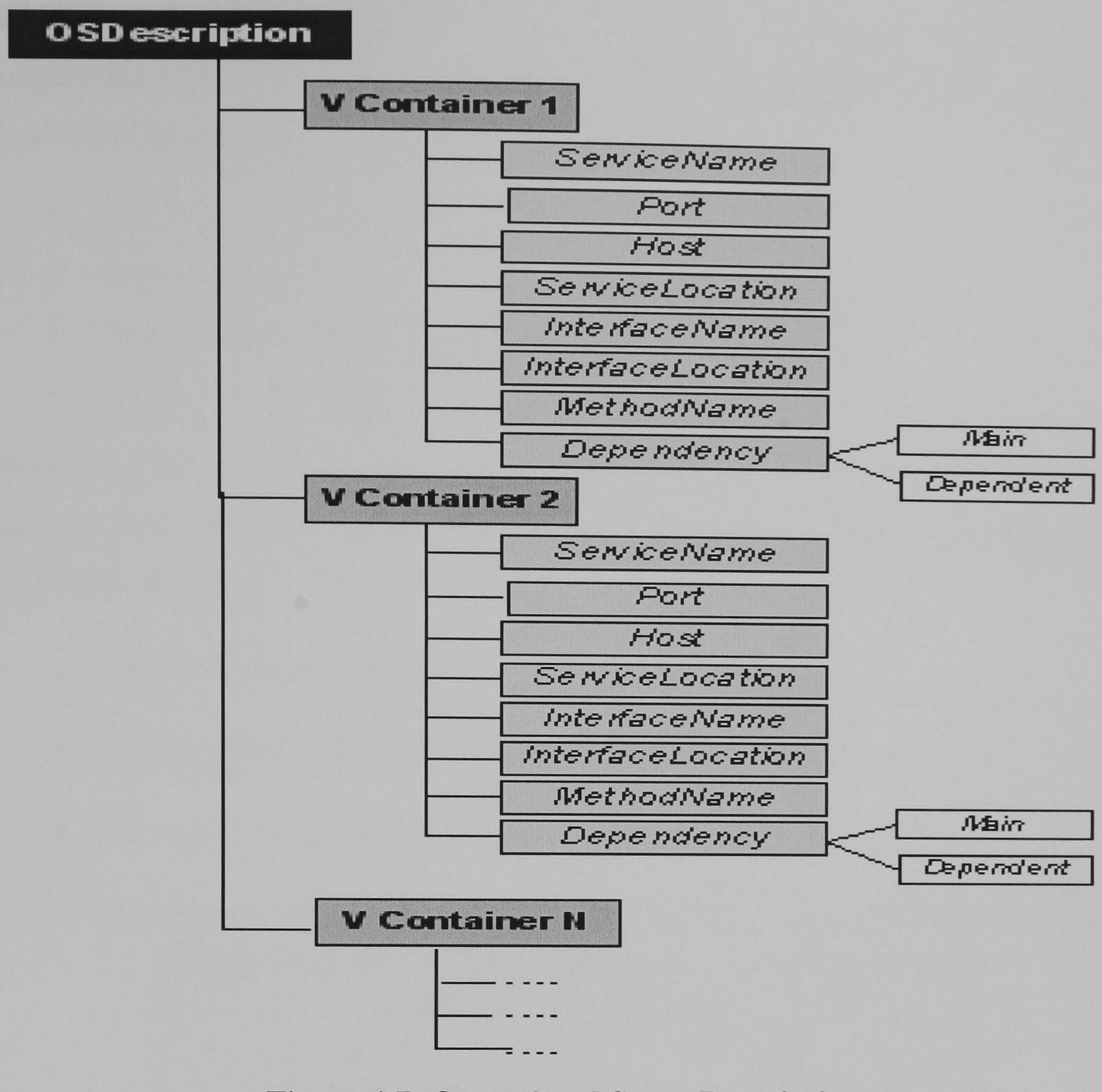



**Figure 4.7: Operational Space Description.**

The operational space is a virtual space where the services run. It can contain one or more virtual containers. The container is the virtual environment where groups of services are executed and live during their life cycle. Each container can contain one or more services. As illustrated in Figure 4.7, the service description itself is based on the service description model that was described at the beginning of this section.

All of these description models were implemented using an XML format. The description of implementation is provided in Chapter 6. The next section describes the self-healing capabilities of the system, and how the OSAD model makes use of this behaviour.

## 4.5 Self-Healing in the OSAD Model

Self-healing systems must have the ability to modify their behaviour in response to changes in their environment, such as resource variability, changing users' needs,

mobility and systems faults [81]. In our model, we concentrate on system fault responding behaviour. According to Oreizy *et al.* [45], the lifecycle of self-healing systems must consist of four major activities.

- Monitoring the system at runtime
- Planning the changes
- Deploying the change descriptions
- Enacting the changes

Based on these activities and the existing architecture of the OSAD model we identified and implemented the self-healing behaviour in our model as the set of the following activities:

1. Monitoring the application
2. Failure/change planning (not implemented)
3. Discovery of alternative component.
4. Implementing the change/ replacement of failed component with alternatives

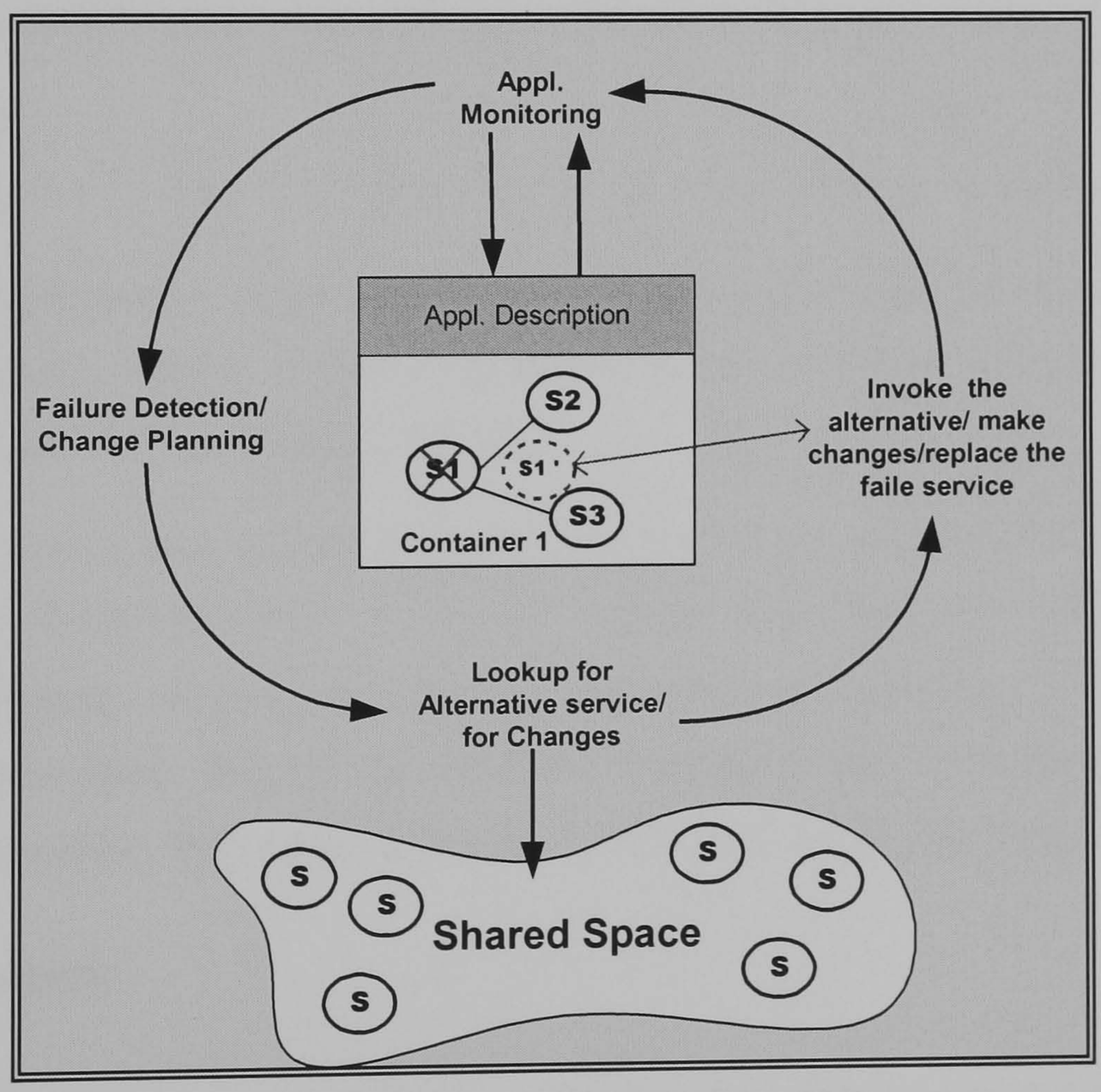



**Figure 4.8: The lifecycle of self-healing behaviour in OSAD**

The lifecycle of self-healing behaviour in the OSAD model, according to these activities, is demonstrated in Figure 4.8.

Having a defined place such as virtual container for a group of services, makes monitoring possible. As the new application is running in a virtual container, monitoring is required to detect changes or failure. Initially, the monitoring system obtains information about each container. The information is generated after the application lifecycle starts in a particular container and contains information about the location of the application, the number of the services that make this particular application and information about the dependencies between services. Having this type of application description allows the monitoring system to continue monitoring the container. As soon as one of the components of the application fails the system starts the recovery operation. Knowing the name of the failed component, the system starts looking for a replacement and the changes are planned. After the lookup and discovery of alternative components and knowing the location of the failed component, the system installs the newly discovered component and replaces the failed one. Consequently, the system performs recovery during runtime, "on the fly". In other words, the self-healing behaviour of the system is activated.

These functionalities are encapsulated in the *System Manager*, the one of the framework components. The *System Manager* contains two sub-services:

- **Monitoring:** is the service that is responsible for monitoring new applications to detect failure. It should be able to identify the failed service and send a request for an alternative service.
- **Recovery and/or Reconfiguration:** is the service is responsible for notifying the Assembly Service of a new request. From the new requirement, the assembly service can discover an alternative service without human intervention. Part of the recovery process is the reconfiguration of the application. This means that a new service will replace the failed service.

## 4.6 Summary

Distributed applications are difficult to develop, and hard to connect and manage. Furthermore, current software architectures are not yet up to the challenge of a large scale networked, dynamically changing environment. The client/server model is too

static and brittle for the new generation of networked systems. Rather than creating monolithic, client/server applications, there is a need for new technologies, designs and architectures that will enable developers to create systems with new functionalities.

This work explores services in relation to the concept of software components, and describes how current component-based development practices provide a tiered foundation for the implementation of a service-oriented architecture. The design model of the system described in this chapter is built using both service-oriented and component based development concepts.

The OSAD model encapsulates a number of functionalities such as service discovery, registration, lookup, invocation and assembly. Furthermore, it is designed and implemented with a self-healing behaviour that makes the system adaptable and reconfigurable.

# Chapter 5

# The Architecture

## 5.1 Introduction

Current software architectures are not up to the challenge of a highly networked, dynamically changing environment. The client/server model prevalent today is too static and brittle. Typically, clients only communicate with the servers they were originally written for, using specific protocols, under specific deployment conditions. The interface between client and server is private, leading to a collection of closed "stovepipe" systems. Most of these implementations assume that the underlying network is static, reliable and trustworthy. However, with the advent of Personal Area Networks, such as Bluetooth, the network is very dynamic. As a person moves the set of devices and services visible to them changes. The network revolution has left software lagging behind, however, a service-oriented, discovery-based component architecture [30] is anticipated to provide a solution for new networked autonomic systems.

In this chapter we describe the overall architecture of our system, which will be followed by the architecture model of the On-demand Service Assembly and Delivery (OSAD) (Sec. 5.3).

## 5.2 The Big Picture

As illustrated in Figure 5.1, OSAD is a service-oriented architecture, which is, for convenience organised in a three-layered model;

- First layer represents the application development support environment that is a collection of *Platform Services*.

- Second layer, called application supporting services and tools, represents all services and tools that were developed as the Impromptu *Framework Component Services.*

- Third layer is the representation of what the framework offers. The third layer is called the *Application Layer* and shows the assemblage of distributed services. In other words, it shows the federations of distributed services. Each layer is described in detail below.

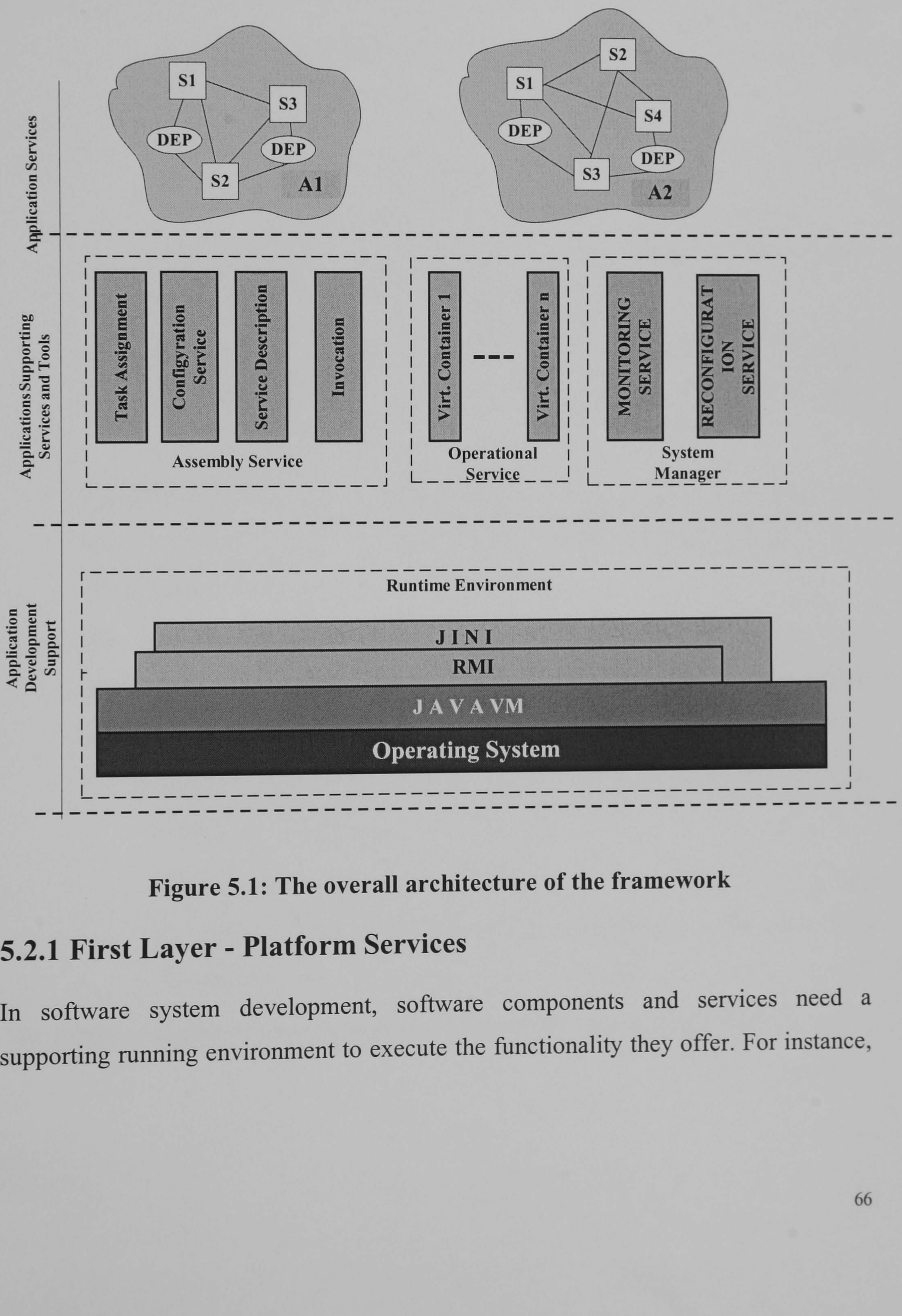



**Figure 5.1: The overall architecture of the framework**

## 5.2.1 First Layer - Platform Services

In software system development, software components and services need a supporting running environment to execute the functionality they offer. For instance,

as shown in Figure 5.1 the Impromptu framework requires a network layer, OS layer and a Java Virtual Machine [82] and Jini core services [83] (Sec. 2.3.2).

### 5.2.2 Second Layer – Framework Components

The second layer represents framework components. The core components of the system are:

- **Assembly Service** that is a combination of sub services including;

  *Task Assignment* – enables the user to assign the task to the system. Using this service, the user will be able to submit the query for the desired services that will make the application.

  *Configuration Service* - places application services into containers and auto-generates application service proxies.

  *Service Description Model* – enables the client side (human or system) to get the method signatures of the service.

  *Service Invocation* – based on the method signatures that client gets from the service meta-description; this service performs the actual invocation of the service.

- **Operational service:** is a combination of virtual containers where the services are executed. This service maintains the support environment required by certain application services.

- **The System Manager:** is responsible for monitoring the applications, detecting failure and performing the recovery. Therefore the system manager contains two sub-services:

  *Monitoring service* – monitors the applications in different containers and detects a failure

  *Recovery/Reconfiguration* – performs the recovery based on the discovery of an alternative service and replacing the failed service with the new one.

The details of implementation of these services are provided in Chapter 6.

### 5.2.3 Third Layer – Distributed Applications and Application Services

The third layer represents the applications (assemblies) developed using the framework. Thus, as shown in Figure 5.1, this layer contains the Impromptu distributed application. As defined by Emmerich, [36], a distributed application is:

> *"... 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".*

Based on this and many other service-oriented distributed application definitions offered in literature, we describe distributed applications as:

> *"A collection of application services spread over networked computers, which clients use remotely via middleware services" where "Application services represent physical and logical concepts such as a printer (hardware service) and a calculator, text editor (software service)."*

## 5.3 The Architecture of the OSAD Model

The On-Demand Service Assembly and Delivery (OSAD) architecture has been designed based on emerging design principles from, for instance; service-oriented architectures, distributed object systems and component-based middleware (Fig. 5.2).

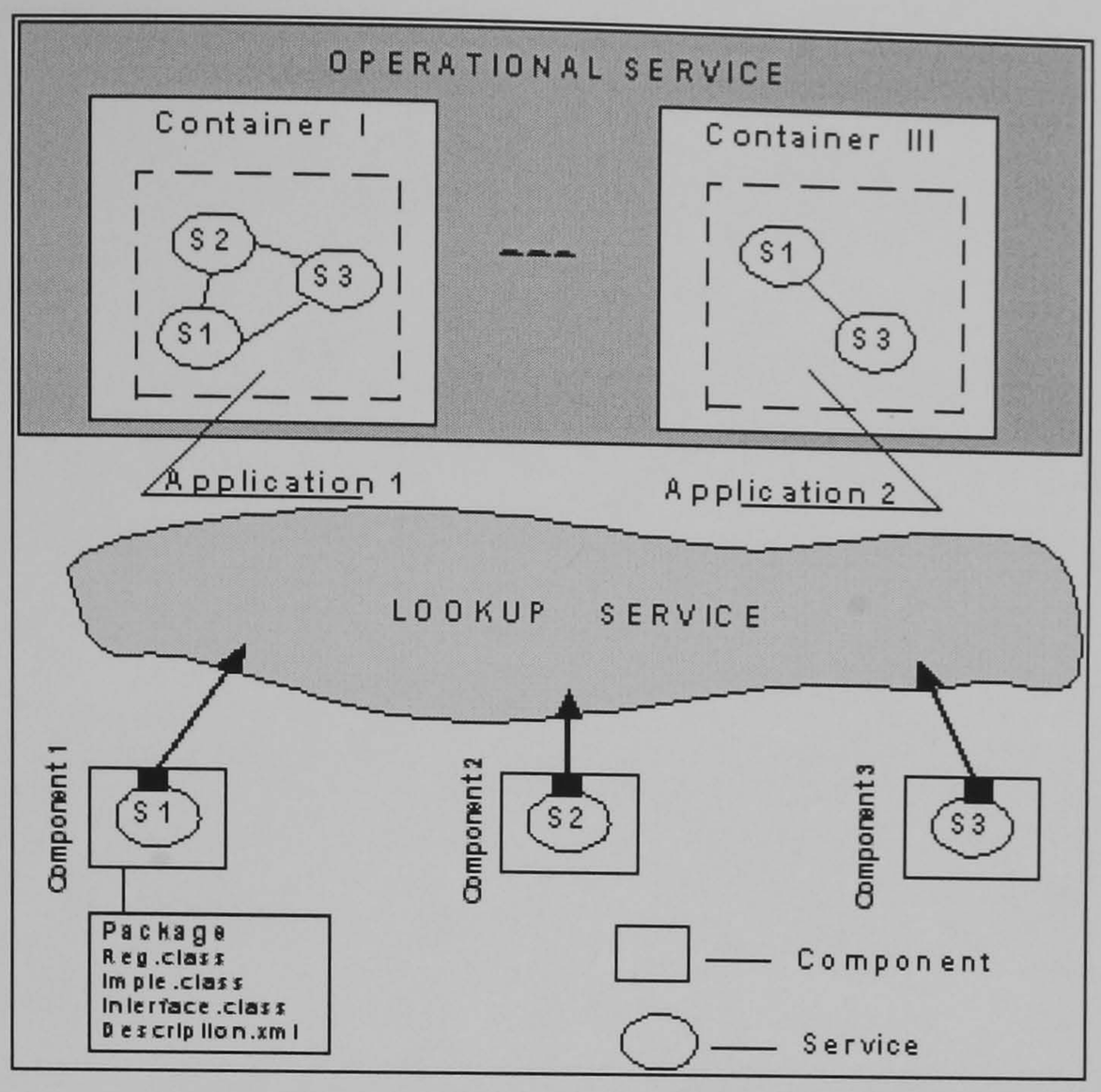



**Figure 5.2: OSAD Architecture**

The *Operational Service* is a Virtual Space containing one or more *Service containers*, where each is a virtual environment in which one or more services are assembled (composed) ready for incorporation into a new application. A *Component* is the set of all necessary class files for a given software component together with the required implementation of service publication, discovery and service invocation via RMI. Components are typically stored in a Java archive (Jar) files.

A *Service* is a logical concept that may be a hardware service, a software service or a combination of the two. A Service can be implemented and provided by a specific component for use by any other component. Services may be distributed across several different machines and accessed through Jini's discovery/lookup mechanism and used through Java's RMI protocol. Discovery allows Service-Oriented applications to dynamically find and publish services. Discovery is a core concept that allows service-oriented applications to adapt to their environment. The implementation of this architectural model is detailed in Chapter 6.

## 5.6 Summary

The client-server architectures of the last decade are simply not flexible or scalable enough to build distributed services and applications that satisfy users' needs in this

new era of networking. Instead, researchers in academia and industry are looking to service-oriented architectures to solve design problems. This chapter described the service-oriented architecture designed and implemented (Chap. 6) to address the problems existing in this area.

The overall architecture of the framework was presented as a three-layered model, containing: (1) the layer of platform services supporting the development and maintenance of framework services; (2) the layer of framework component services and (3) the layer of distributed applications and application services.

The architecture of the On-demand Service Delivery and Assembly (OSAD) model was also described in this chapter. The OSAD model is a representation and union of the framework component service, which was described as the second layer of the overall architecture.

The following chapter presents an in-depth description of the implementation issues associated with the second layer of the above described three layered architecture, including an in depth description of each component and service of the framework

# Chapter 6

# Implementation of IMPROMPTU Framework

## 6.1 Introduction

In this chapter we describe the prototype implementation of the On-Demand Service Assembly and Delivery (OSAD) model presented in Chapter 4. The prototype is called "*Impromptu*". The OSAD model aims to organize the following issues in a uniform framework: (1) on-demand service delivery and invocation regardless of the location of the service, (2) the automatic assembly of the application in an ad-hoc manner based on the user's requirements, and (3) the ability of self-healing at runtime in terms of replacing the failed component of an impromptu application.

The Impromptu system was written in the Java programming language [82]. The choice of the implementation language is based on a number of reasons. The mechanisms provided in Java such as dynamic class loading, object serialization, remote method invocation and reflection allow the flexibility required for runtime adaptation and invocation. Also the wide deployment of JVM is appropriate for experimental and evaluation purposes.

This chapter will provide a detailed description of the three core services of the Impromptu Framework namely:

- *Assembly service,* which implements the first two concepts of the OSAD model - on demand service assembly and delivery.
- *Operational service,* which provides a virtual platform for *Impromptu* services.
- *System Manager Service,* which brings the self-healing behaviour to the framework.

In addition, this chapter will provide a description of two example applications namely: Home Appliances and Software Services scenarios.

## 6.2 Implementation Requirements

An Impromptu application is a combination of distributed services provided by distributed components over a local or global network. A component in a distributed application is a self-contained binary implementation, which consists of one or more objects. These objects occur as instances of the classes that make 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.

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 [3]. 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 the developed framework, which provides a set of services that can be used to discover, assemble and manage the application services of a distributed application.

### 6.2.1 The Choice of Jini Technology

As a proof-of-concept, Java and Jini middleware were chosen for the implementation of the proposed framework and associated architecture. Essentially, Jini enables distributed applications to be developed as a series of clients that interact with application services. Clients use stubs or proxies, implemented as Java interfaces, to remotely invoke application service methods (specified by the proxies) relying on Java's RMI to route the invocations over a network. Jini applications consist of a federation of application services and middleware services (e.g. lookup) and a series of clients and Java Virtual Machines (JVMs) distributed across several computing platforms.

Moreover Jini was chosen due to its rich support for service-oriented development and support for a variety of services that may be hardware-based, software-based or a

combination of both. Jini does present some limitations, the main being its Java dependency, whereas technologies such as CORBA [16] and DCOM [17] offer greater support for development in heterogeneous environments through their language independence. However, with some extra effort, Jini may also support heterogeneous components through Java's native code methods, which act as wrappers around non-Java objects.

## 6.3 IMPROMPTU Component Services

The Impromptu system described in this chapter is an implementation of the On-Demand Service Assembly and Delivery Model (OSAD). As described in Chapter 4, the OSAD was developed for distributed applications. A distributed application is the collection of distributed services provided by distributed components. To implement the main objectives of the OSAD model, the prototype component services were developed: (i) Assembly Service; (ii) Operational Service; and (iii) System Manager (Fig. 6.1).

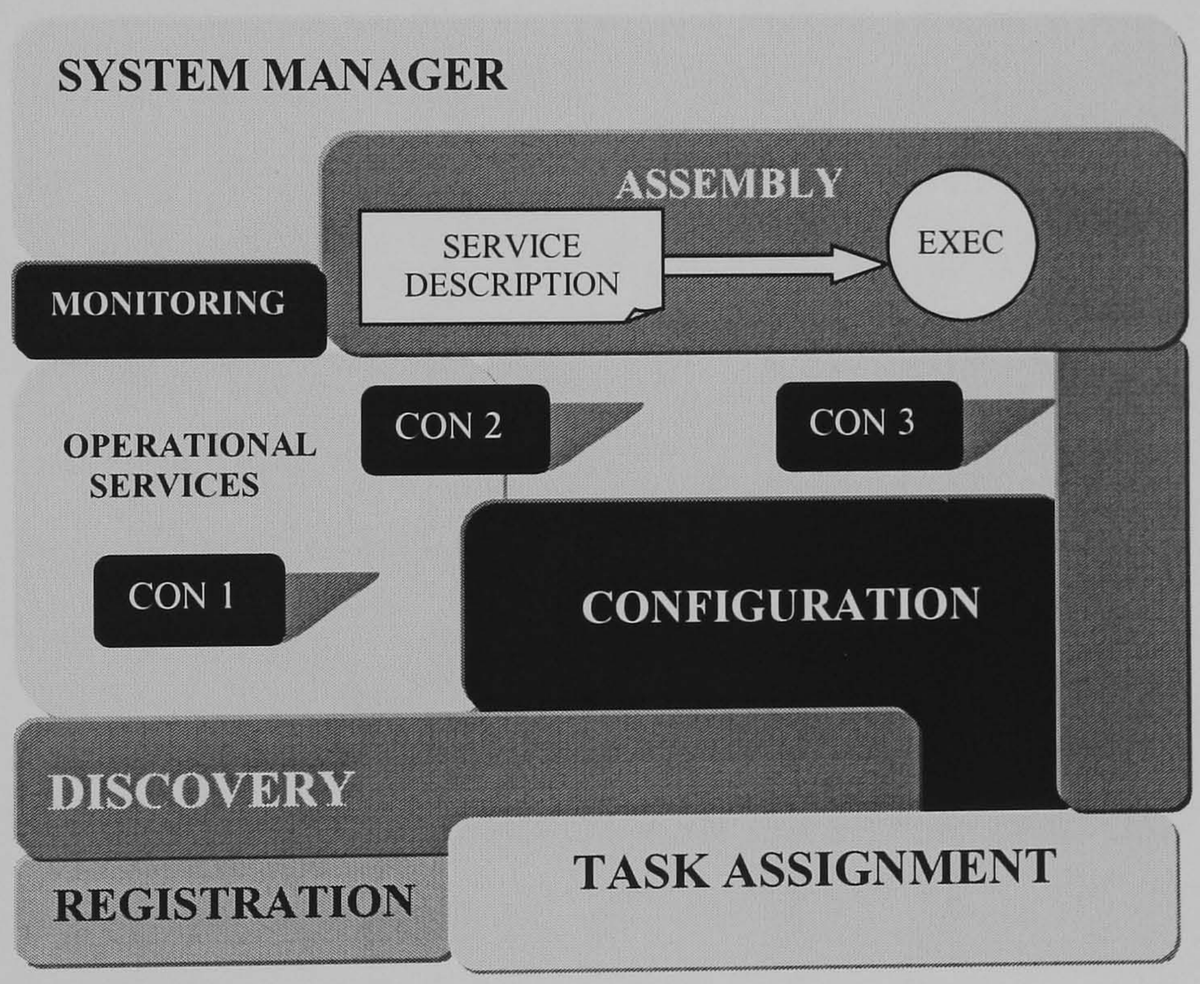



**Figure 6.1: Impromptu Component services**

Each component service consists of sub-services/components (Fig. 6.1). The Assembly Service contains: a *task submission, registration/discovery* and *configuration* services as well as the S*ervice Description* and *Execution* models.

The *Operational Service* supports the system with the environment where the services are executed. In other words, it is a virtual environment containing a number of virtual containers.

Finally, the *System Manager* consists of *Monitoring* and *Recovery/Reconfiguration* services and provides the self-healing behaviour for the system.

The rest of this chapter will provide a detailed description of the implementation of the OSAD model services.

## 6.3 Assembly Service

In line with the service-oriented architecture, each service assembly (or *Impromptu* application) is developed as:

> *"... independent sets of interacting services offering well-defined interfaces to their potential users. Similarly, supporting technology must be available to allow application developers to browse collection of services, select those of interest, and assemble them to form or create the desired functionality " [84]*

Researchers and application developers describe service composition, assembly or aggregation in different ways. In [85] assembly of service is interpreted and implemented as an assembly of Body and Soul components (Sec. 3.3), where Soul components are mapped onto Jini clients and as Body components onto Jini services Body components register their attributes with the directory service when they are instantiated. Soul components make enquires to the directory service in order to obtain identifiers of available Body components necessary to contract an application. In the Rio project [47] (Sec. 3.2), an Operational String describes an aggregated collection of application and/or software assets that when put together provide a specific coarse grained service on the network. The software assets are envisioned as being Java service objects.

In this work, the assembly service is defined as the process of composing a required distributed application as a set of interacting services, offering the desired functionality (e.g. the functionality of a heating system integrated with the functionality of a timer can offer new functionality). At the same time, this work does not claim to perform the assembly of functionality at software component integration

level (e.g. assembling the components of the software developed by different individuals using different technologies). The assembly Service is the implementation of the service assembly approach used in this work.

The *Assembly Service* is the core component of the OSAD model. It was designed and implemented for the on-demand service discovery, invocation and automatic assembly of services in an ad-hoc manner. In the process of creating a distributed application, the user should somehow specify the requirements of the services. The match between users requirements and available on-demand candidate services is determined dynamically at runtime. Creation of the new application can be done by manual configuration or users might like to leave it to the system to configure automatically.

The Assembly Service consists of the following sub services and models:

- **Task submission** – to configure a new application automatically, the system must know about the users requirements. The task submission service enables the user to input requirements. These requirements are saved as an XML document.

- **Registration/discovery** - these are Jini core services but to make them more generic than those offered by Jini, Registration/ Discovery services were modified. Both of these services are separate reusable components of the Assembly service implemented in Java as stand alone classes.

- **Configuration service** – is used for placing the discovered service in the virtual containers, where the service invocation takes place.

- **Service description model** – is used for passing information about the service from the service developer to a client. Service descriptions are documented as an XML document. Each Impromptu service is registered and discovered with an attribute pointing to the location of the description file.

- **Service invocation model** – is based on Java Remote Method Invocation. For service execution, the client side should know the name of the method. To understand the structure of the service is not an easy task, but by

parsing the XML based service description documents it becomes easier for the invocation service to get the name of the invocation method.

The interaction between these components is shown in the UML class diagram in Figure 6.2

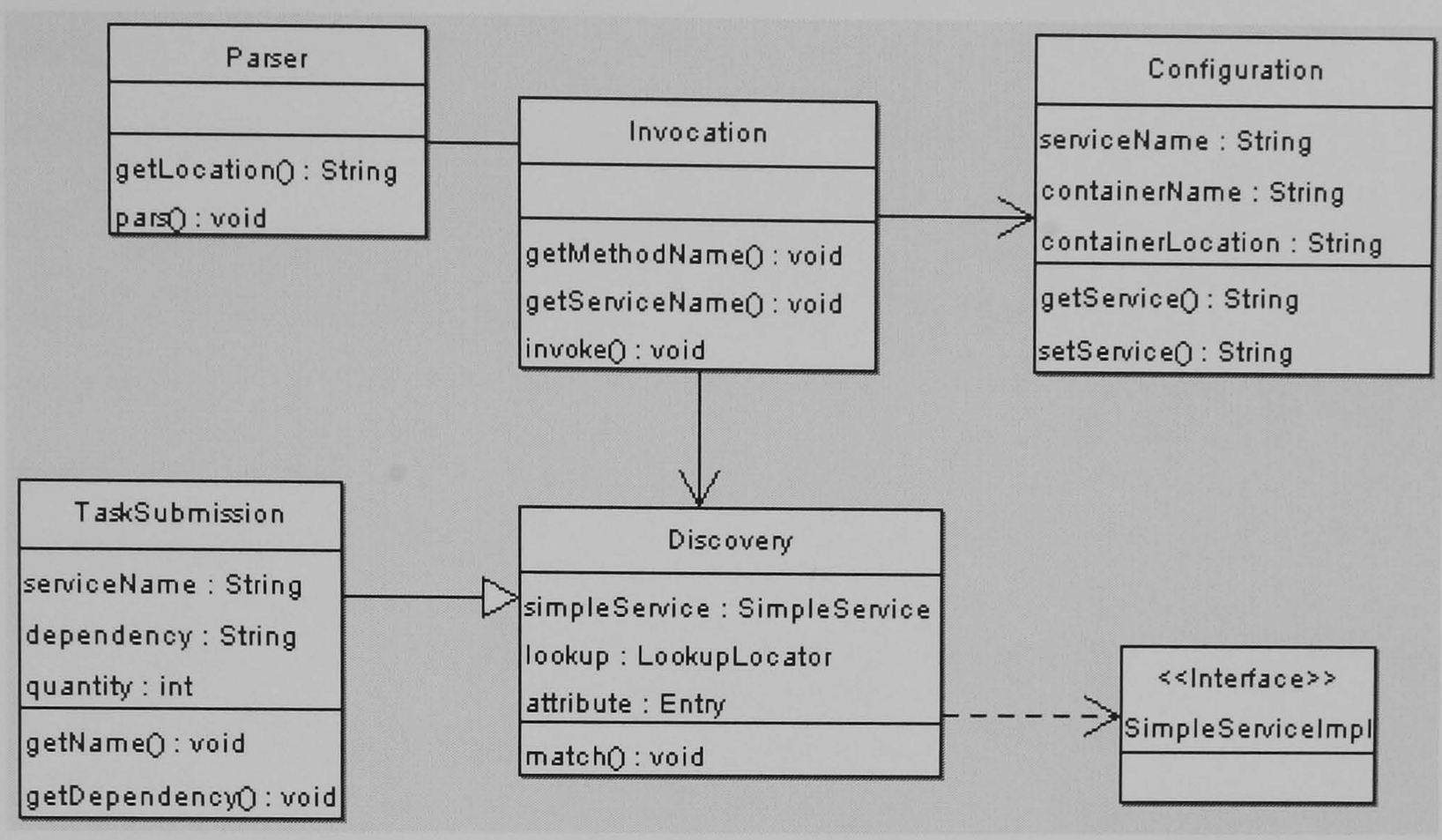



**Figure 6.2: The UML Class Diagram for Assembly Service**

## 6.3.1 Task Submission

The *task submission* sub-service enables the user to submit requests for particular services to the system. In other words, using the *task submission* sub-service, a user can create a desired application scenario.

The implementation of the t*ask submission* sub-service is based on the task description model described in Chapter 4. There are two main attributes a user is required to define. These attributes are:

- Service Name, the attribute that is required for the system to make a service discovery and match.

- Dependency, the attribute that tells the system which service is dependent on which.

Each task scenario described and submitted by the user is formatted as an XML document and saved with a unique version number. These documents are saved so

that a new user has the ability to use it if his requirements match the task defined by another user.

### 6.3.2 Registration/Discovery

The most important feature of the distributed service is to be discoverable. In order to make the service discoverable, it should be registered with some middleware service that gives the opportunity to client services to lookup and locate a desired service.

The registration and discovery sub-services in the Impromptu Framework are the components of the Assembly Service. The implementation of these services is based on Jini registration and discovery services. But for the purposes of making the system generic we separated them from the Jini service and made as independent reusable components.

The service is registered by calling *register()*, which returns a *ServiceRegistration* object. A service can announce a number of entry attributes when it registers itself with a lookup service. Each service can include a number of attributes. It is the choice of the service developer and provider, but the Impromptu service has to have two attributes always defined. These attributes are the name of the service and the location of the service description file. The Impromptu client chooses the appropriate service by the name of the service.

```
public void register() {
     SimpleService simpleService;
     LookupLocator lookup;
     Entry[2] attributes;
     JoinManager joinmanager;
   try {

     attributes = new Entry[1];
     attributes[1] = new Name("SimpleService");
     attributes[2] = new Comment
("http://cmsegris:8080/userdoc/SimpleServiceDiscription.xml");

      simpleService= new SimpleService();
      joinmanager = new JoinManager
  (
    simpleService,
    attributes,
    simpleService,
    new LeaseRenewalManager ()
);
        }catch (Exception e)
         {
         System.out.println("server: MyServer.main(): Exception " + e);
         }
}
```

**Figure 6.3: Implementation of registration sub-service**

Discovery is also based on the Jini discovery service and is used in two slightly different ways. Calling the `matchis.totalMatchis` method the user is able to obtain the complete list of available services that are registered in the network at that particular moment.

The other method is when the assembly service needs to perform a discovery according to user requirements. In this case, the discovery is based on the attribute `Name`. For instance, if the user requires a service called "Calculator", the system will match according to the name and will discover only available calculators.

### 6.3.3 Configuration

Configuration for management purposes consists of setting particular attributes before an execution, as well as defining the policies that drive their utilization. In this work, we have focused on one aspect of the configuration, namely the placement of the services in virtual containers.

The Configuration service is responsible for placing the service in its new location. The new location can be any container that is available at that time. However, there must not be another set of services running in that particular container. The Configuration service also understands the task submitted by the user, so that all services included in the defined task will be placed in the same location. This makes the task of the monitoring service easier, as the monitoring service monitors the containers individually.

### 6.3.4 Service Description

The reason for the service description model development was the limitations inherent in Jini technology. The invocation of the service is based on Java RMI, in most cases the method name is not known on the client side, as only the service provider knows about it. Therefore, there was need for a technique that would make it clear to the client the method of invocation for a discovered service. For this purpose, as described in chapter 4, we developed the Service Description Model.

We implemented this model in XML format. As shown in Figure 4.2 (Chap. 4), each service is described with the following attributes:

- ***Service Name***, a required attribute, providing a human and computer readable identifier for each service.

- ***Service Location,*** the actual physical location of the service.

- ***Interface Name,*** a required attribute, providing a human and computer readable identifier for the service interface. The interface where the methods of invocation are described.

- ***Interface Location,*** the actual physical location of the service interface file.

- ***Method Name,*** a required attribute, providing human and computer readable identifier for service execution.

The service description document for a simple service is presented in the Figure 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Simple Description of Service -->
<!DOCTYPE ServiceDescription SYSTEM "ServiceDescription.dtd">

<ServiceDescription>

  <ServiceName>simpleService</ServiceName>

<ServiceLocation>http://cmsegris:8080/userdoc/SimpleService</Servi
ceLocation>
  <InterfaceName>SimpleServiceInterface</InterfaceName>

<InterfaceLocation>http://cmsegris:8080/userdoc/SimpleServiceInter
face</InterfaceLocation>
  <MethodName>seyHello</MethodName>

</ServiceDescription>
```

**Figure 6.4: Simple service description**

### 6.3.5 Service Invocation

Service Invocation takes place after all the services requested by the user are discovered and placed by the configuration service in a suitable virtual container.

Service invocation was implemented using Java RMI. Each discovered service has an additional attribute pointing to the location of the service description file.

```
Entry entries = matches.items[1].attributeSets;
String entry1_name = entries[1].toString();
String entry2_locatio1 = entries[2].toString();
```

**Figure 6.5 Service Attributes**

The entry2_location parameter is passed to the Parser.class and the service description document is parsed (Document document = builder.parse(new File(entry2_location))).

The `parser.class` passes the following parameter to the `invocation.class`.

```
NodeList mNameNodes =
                ServiceDescriptionNode.getElementsByTagName(
"MethodName" );

      if ( mNameNodes.getLength() != 0 ) {

      Node MethodName = mNameNodes.item( 0 );

String methodOfInvocation = MethodName.getChildNodes().item(0
).toString());
    }
     else
          Node ServiceLocation = mSLocationNodes.item( 0 );

String serviceLocation = MethodName.getChildNodes().item( 0
).toString());
```

**Figure 6.6: Parser**

After getting the `methodOfInvocation` parameter, the Invocation class invokes the service. This describes the simplest case, when each service offers only one function. An example of such simple services are networked appliances such as a printer, scanner, or home appliances such as networked clock, heating system, camera and so on.

```
public class GetMethod() {

..

 System.setProperty("java.rmi.server.codebase",
                    "http://cmsegris:8081/");

..
 Class c = Class.forName(serviceLocation);
 System.out.println("Class " + c.toString());

..
Method m[] = c.getDeclaredMethods();
 for (int i = 0; i<m.length; i++)
 String methodName = m[i].toString();
```

**Figure 6.7 Service Location**

A distributed service can also be complicated, such as a software service (e.g. calculator). In this case, the invocation can be implemented in different ways. If the service invocation method name is not known, in other words - `if (mNameNodes.getLength() = 0)` then the parameter that the parser class passes is the location of the service class file.

The name of the invocation method can then be obtained by using the Java reflection method. The `serviceLocation` parameter is passed in the same way to the invocation class as in simple case described earlier.

## 6.4 Operational Environment

The Operational Service maintains the support environment required by certain application services. The operational service is a virtual environment containing a number of virtual containers. This is the environment where services are executed after the system performs discovery. The point of having a number of virtual containers is that a user can have a different assemblage on different containers at the same time. When a service is executed, a remote message is sent to the monitoring centre with the description the location of the service. This information is documented as an XML document. When the *System Manager* detects a failure, a check can be made on what was running in that particular container and plan a strategy for recovery.

## 6.5 System Manager

Application management is based on the use of a combination of a middleware-based instrumentation monitor and control services to monitor runtime behaviour and adapt behaviour in response to runtime conflicts. Controller concepts and control services have recently gained popularity amongst the self-adaptive software community, typified by [86], who use control services to adapt the structural configuration and dynamic behaviour of an application. Structural components can evaluate their behaviour and environment against their specified goals with capabilities to revise their structure and behaviour accordingly.

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 a distributed application, but only with limited success, because of the disturbance and possible shut down of the whole system. For this reason, dynamic autonomic management, applied at runtime, has recently attracted the attention of researchers concerned with distributed applications development and management [87].

Described by [8], the system manager consists of three main processes to control service behaviour: report the notification of failure, find an appropriate alternative and to recover the system from failure. These processes are specified as: a *Service Monitor*, the *Service Diagnosis* and a *Service Self-Repair*.

In the process of the implementation of the system manager, all of the processes described above are important. But as the work described in this thesis is aimed at the development of a framework for on-demand service delivery and assembly with self-healing behaviour, the concentration on developing the system manager was relatively low. For the purpose of detecting failure in one of the services of the Impromptu application, the Monitoring service was developed and implemented. Without identification of the reason or course of conflict, the system manager is able to recover the system from that conflict. For this purpose, the recovery and reconfiguration services were developed and implemented. In this stage of developing this work, the main goal of the work was to test the reaction of the system if the

conflict was detected. Is it able to bring the assembly service, namely discovery and invocation services, into action again in order to perform recovery.

### 6.5.1 Monitoring

As described in [88], the monitoring service receives messages such as remote events concerning failure and/or conflict from an instrumentation service [43] (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.

For the purpose of this work, the monitoring service carries slightly less responsibility than in [43]. The monitoring service, in the Impromptu prototype monitors the operational service and consequently each virtual container is monitored.

The monitoring service is active during the whole life cycle of the Impromptu application. When a service is invoked, the remote event is sent to the monitoring service with the parameter showing the name and location of the service. Location in this case means the location in the operational environment; for instance, service 1 is running in container 1.

In case of manual configuration of a new application by the user, he/she decides if all of the required services are found and invoked and an XML based description document is generated about the current state of the operational environment essentially which container contains what services.
There can also be a case of dynamic configuration. A user can specify the requirements and assign a task to the system. The system finds requested services automatically, instead of the user doing it manually.

In both cases, the output of the system is a description of the current state of the applications. The descriptions are produced for further use by the System Manager, described in [8], for detecting failure and planning recovery. An example of a description document generated by the monitoring service is shown in Figure 6.8.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!--
 New Application Description
  -->
  <!DOCTYPE NewApplication (View Source for full doctype...)>
= <Container1>
  = <Service ServiceID="s1">
      <ServiceName>Calculator</ServiceName>
      <Port>4061</Port>
      <Host>cmsegris</Host>

        <ServiceLocation>http://cmsegris:8080/userdoc/Calc
        ulator</ServiceLocation>
      <InterfaceName>CalcInterface</InterfaceName>

        <InterfaceLocation>http://cmsegris:8080/userdoc/Ca
        lcInterface</InterfaceLocation>
      <Method>getCalcInterface</Method>
      <Dependency>Antecedent</Dependency>
    </Service>
  = <Service ServiceID="s2">
      <ServiceName>DialogWindow</ServiceName>
      <Port>4061</Port>
      <Host>cmsegris</Host>

        <ServiceLocation>http://cmsegris:8080/userdoc/Dial
        ogWindow</ServiceLocation>
      <InterfaceName>DWIn</InterfaceName>

        <InterfaceLocation>http://cmsegris:8080/userdoc/Di
        alogWindow</InterfaceLocation>
      <Method>getDWInt</Method>
      <Dependency>Dependent</Dependency>
    </Service>
  </Container1>
```

**Figure 6.8: Description of the state of container 1**

## 6.5.2 Recovery and Reconfiguration

As earlier described, this work does not address software instrumentation based monitoring, diagnosis and recovery planning issues, but provides a systematic scheme in the case of a service failure.

After a conflict is identified by the control mechanism, a resolution is made for recovery. The recovery process involves:

1. Notifying the assembly service about the resolution. Sending a remote event notification with the request for a new service to be discovered .
2. The Assembly service repeats the actions of discovery and invocation.
3. Reconfiguration of the application takes place. The new service replaces of the failed one.

## 6.6 Illustrative Example Applications

In the following sections, we will describe two example applications that were designed and implemented using Jini middleware with Impromptu Framework services. These prototypes were developed to illustrate the use of the framework and at the same time, test and evaluate the various features of the developed framework. In order to test different use of the framework services, we have developed two different example applications namely: the home appliances and software services example applications. Each application will be described in the following sections. The applications were developed using Java JDK 1.4 and Jini 1.1 and tested on a Windows XP platform.

### 6.6.1 Application 1: Home Appliances

More and more researchers are getting involved in the area of networked appliances as mobile devices become more and more ubiquitous through the use of wireless connectivity. As they connect to wide or home networks, then the easier it becomes for users to utilise the services such networks provide. Moreover, these devices are smart enough to understand each other, to communicate and share information in a networked environment [89]. We have used a typical example scenario of networked appliances that is widely used in the area of distributed, networked services [90] [91].

The application described in this section is a basic example of the initial implementation and use of the Impromptu Framework. Here the home appliances scenario is used to demonstrate the use of the Assembly Service and the System Manager with its self-healing feature.

Before describing the application scenario, it is necessary to give a definition of the network appliances we adopted from existing work and extended it for the purpose of our application. A Networked Appliance is:

> *"...a device that has the capabilities of delivering certain services that have been specified by the provider and can be accessible via a network."*

The application scenario is as follow:

*"The user has home appliances connected to the web. Before she leaves work, remotely from her office she decides to assemble an Impromptu application in order to make her appliances work together (Fig. 6.9). Using the registration service she registers the service in order to make them interactive and discoverable. She knows she needs to wake up at 8 am and have breakfast before leaving home. So she needs a clock, lamp, toaster and a kettle to work together. The new application should be monitored and managed by the system."*

As there is dependency between these services, if the clock fails then the whole application will go wrong. In this case, the system takes care of this federation of services and without user intervention, (the user is supposed to be sleeping) finds another clock on the lookup service and replaces the failed one. As a result, the system makes changes and reconfigures itself.

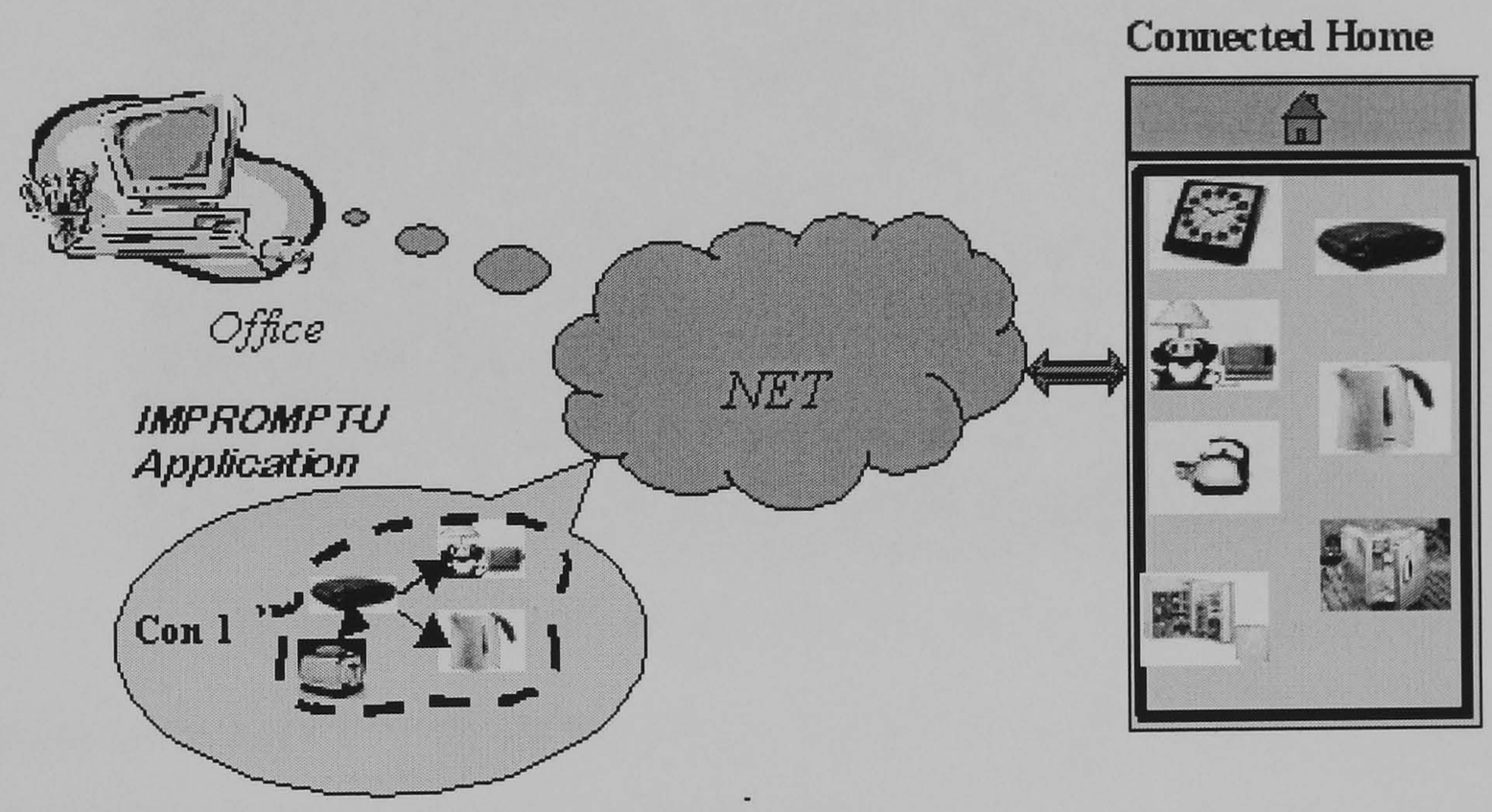



**Figure 6.9: Home Appliances scenario**

By implementing this scenario we have tested the following services of the framework:

- The Assembly service, including the Registration, discovery and invocation sub-services, which provide the functions for finding and invoking services that are available in the network in that particular moment. Also for service invocation purposes, the additional service description documents were developed and tested.

- The System Manager, including Monitoring and Recovery/Reconfiguration sub-services, monitor the applications in order to detect a failure and through the recovery/reconfiguration sub-service, implements the self-healing feature of the system.
- The Graphical user Interface (GUI) of a prototype application is shown in Figure 6.11. The application itself can be activated and run in a following manner:

Run the Jini core middleware services including: RMDI, Web Server and the Reggie services (Fig. 6.10).

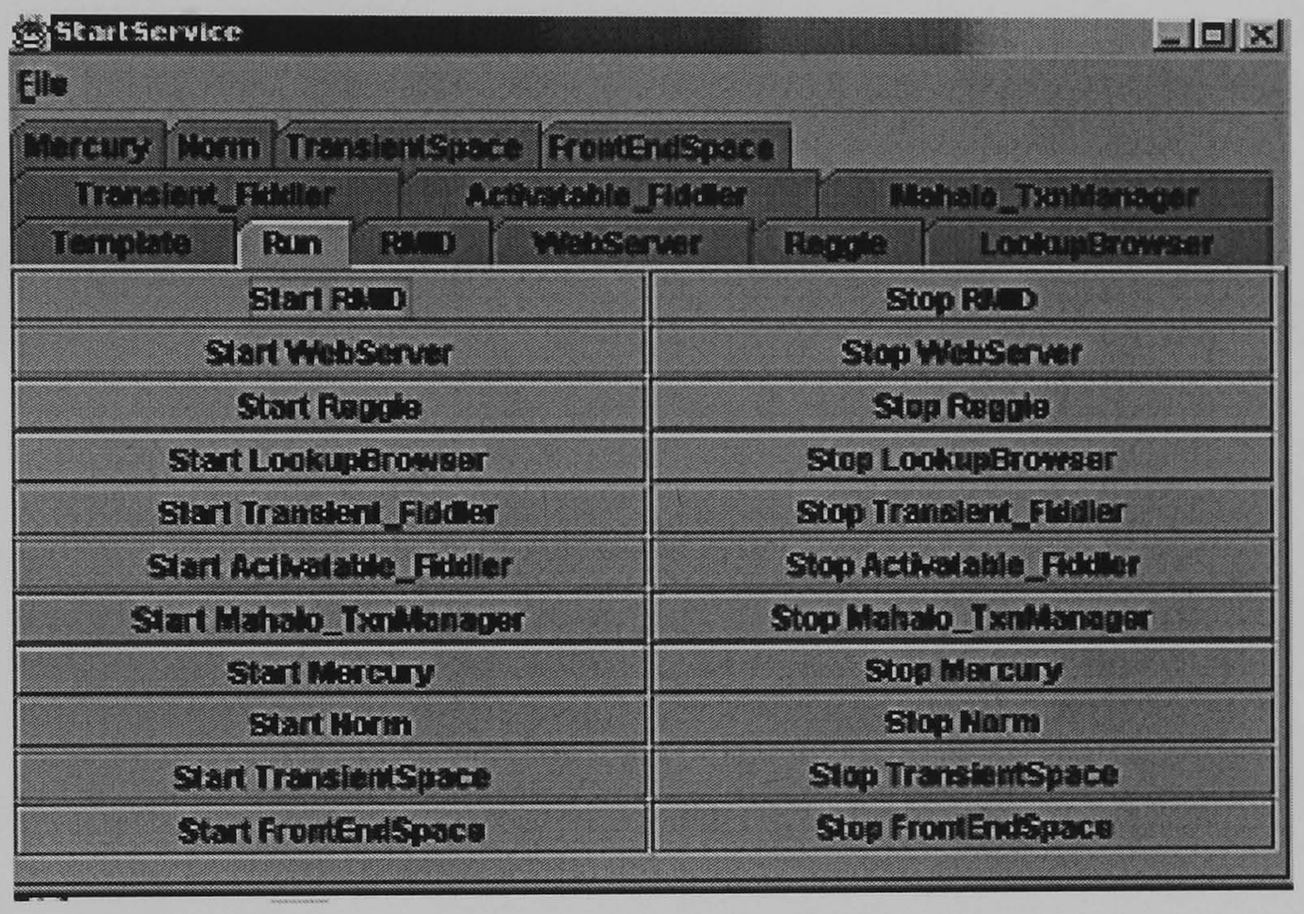



**Figure 6.10: The Jini StartService Application [22].**

As shown in Figure 6.11, all framework services can be executed from the main GUI, which contains three main sections (Fig. 6.11):

- The Service tree; showing all the services that were developed for the evaluation the framework. These services were developed to implement different application scenarios. This means that they are Jini services with additional XML descriptional documents attached to them. By selecting each service and activating the registration button (bottom of the service tree) chosen services can be registered with the Lookup Service.
- The toolbar; provides Icon-based access to all home appliances service registered and available on the home network.

- The Virtual Container; providing container utilities for services assemblies including execution and life-time management.

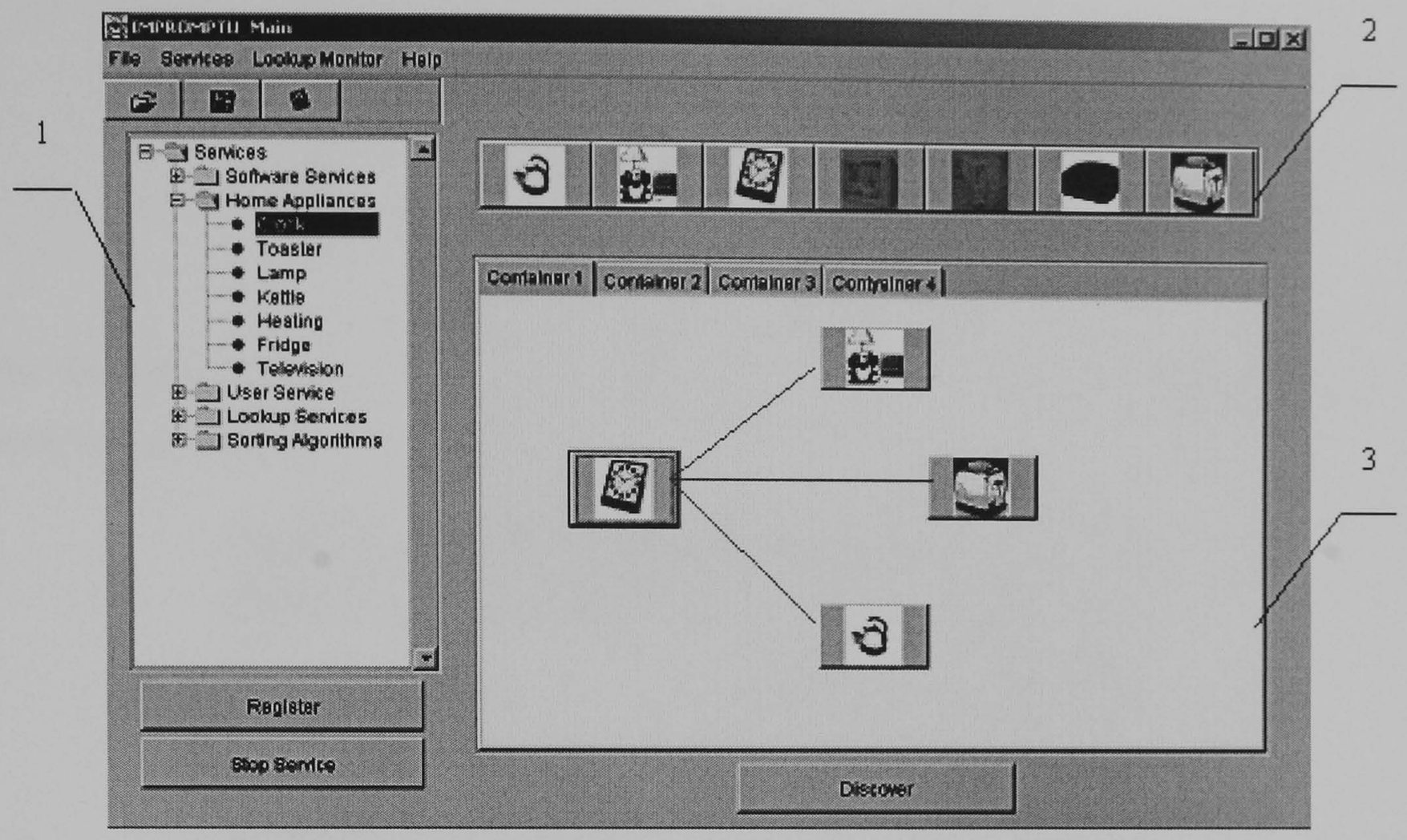



**Figure 6.11: The Main GUI of the example application**

As shown in Figure 6.12, for simplicity, the definition of service dependency including execution order, is done manually through the graphical user interface. However, other approaches as described by Reilly et al. [43] can be achieved automatically through the use of a Dependency Graph Service, which makes use of Java Reflection API.

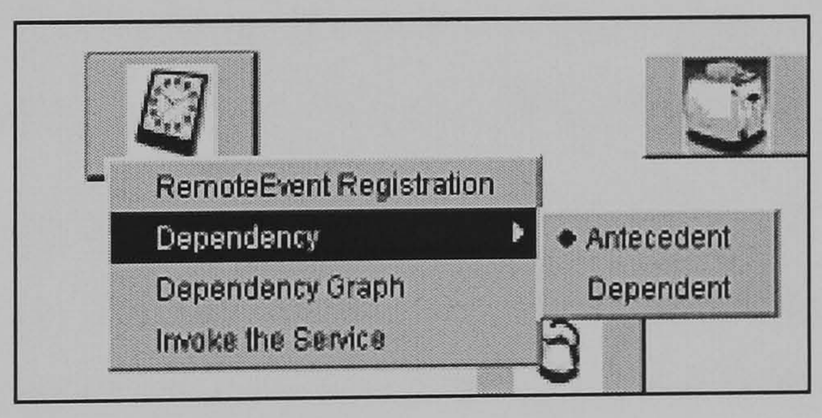



**Figure 6.12: Defining the dependences**

A service can be manually (or automatically) activated through service execution. This is done via the GUI by the "*Invoke the Service*" (Fig. 6.13).

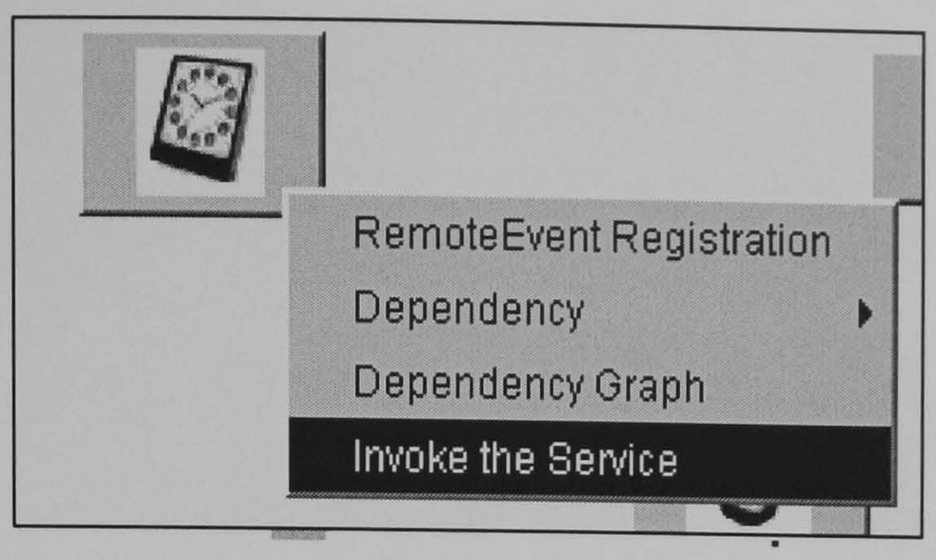



**Figure 6.13: Service Invocation**

The Monitoring service is used to monitor and display the list of running services in each virtual container (Fig. 6.14).

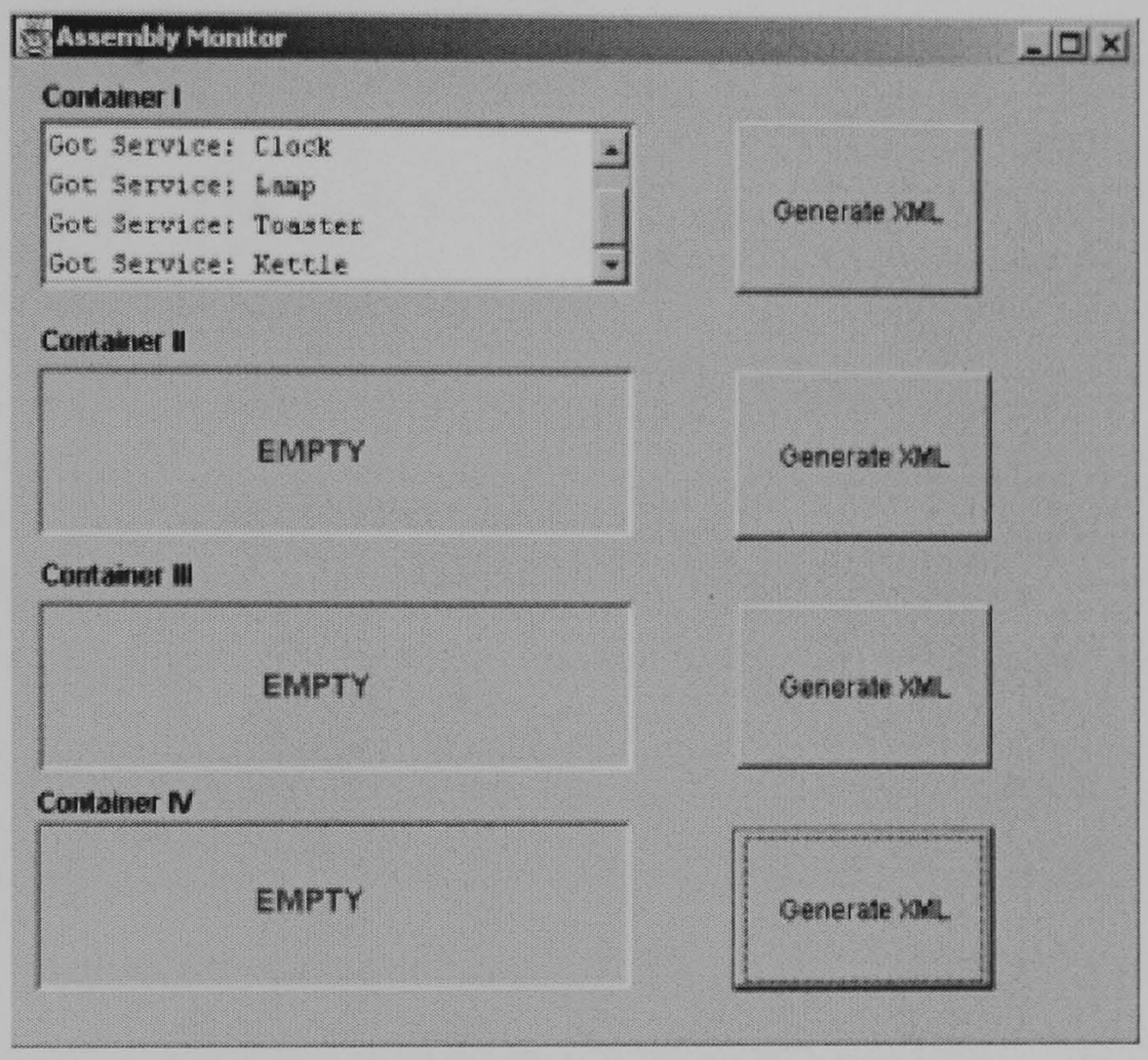



**Figure 6.14: the GUI for Monitoring Service**

If the monitoring service detects a failure of one of the services, for instance, the clock has failed, then the system starts recovery by sending a remote event to the assembly service in order to rediscover and re-invoke the failed service (Fig. 6.15).

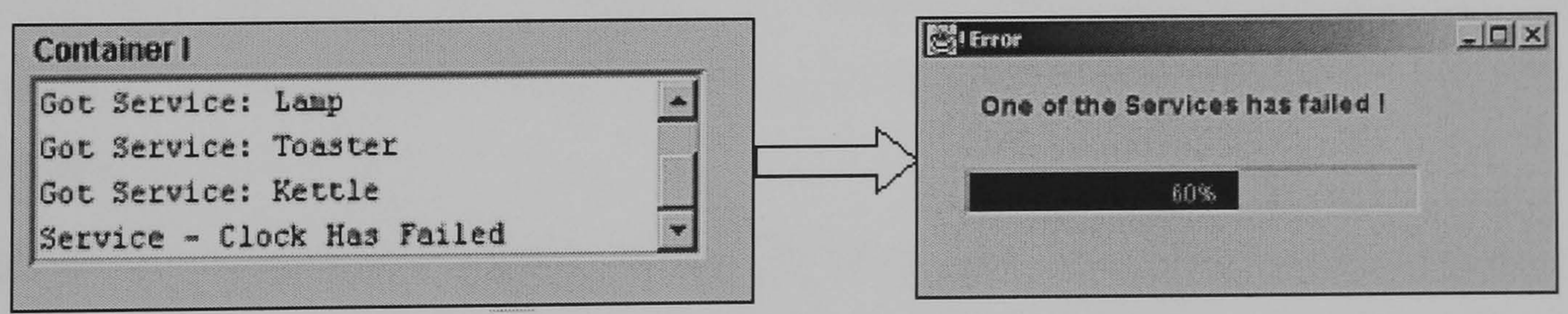



**Figure 6.15: Service failure detection**

All of the steps described above for example application to run and function could be done automatically by assigning a task to the system. For this purpose, we have created an additional GUI (Fig. 6.16), where the number of services can be chosen from a drop down menu. The names of the services can be entered into text boxes. By submitting the requirements, the system will automatically look for the services, generate the XML document and will invoke the services and place them in an available virtual container. The monitoring and recovery tasks will be performed in the same way as described above.

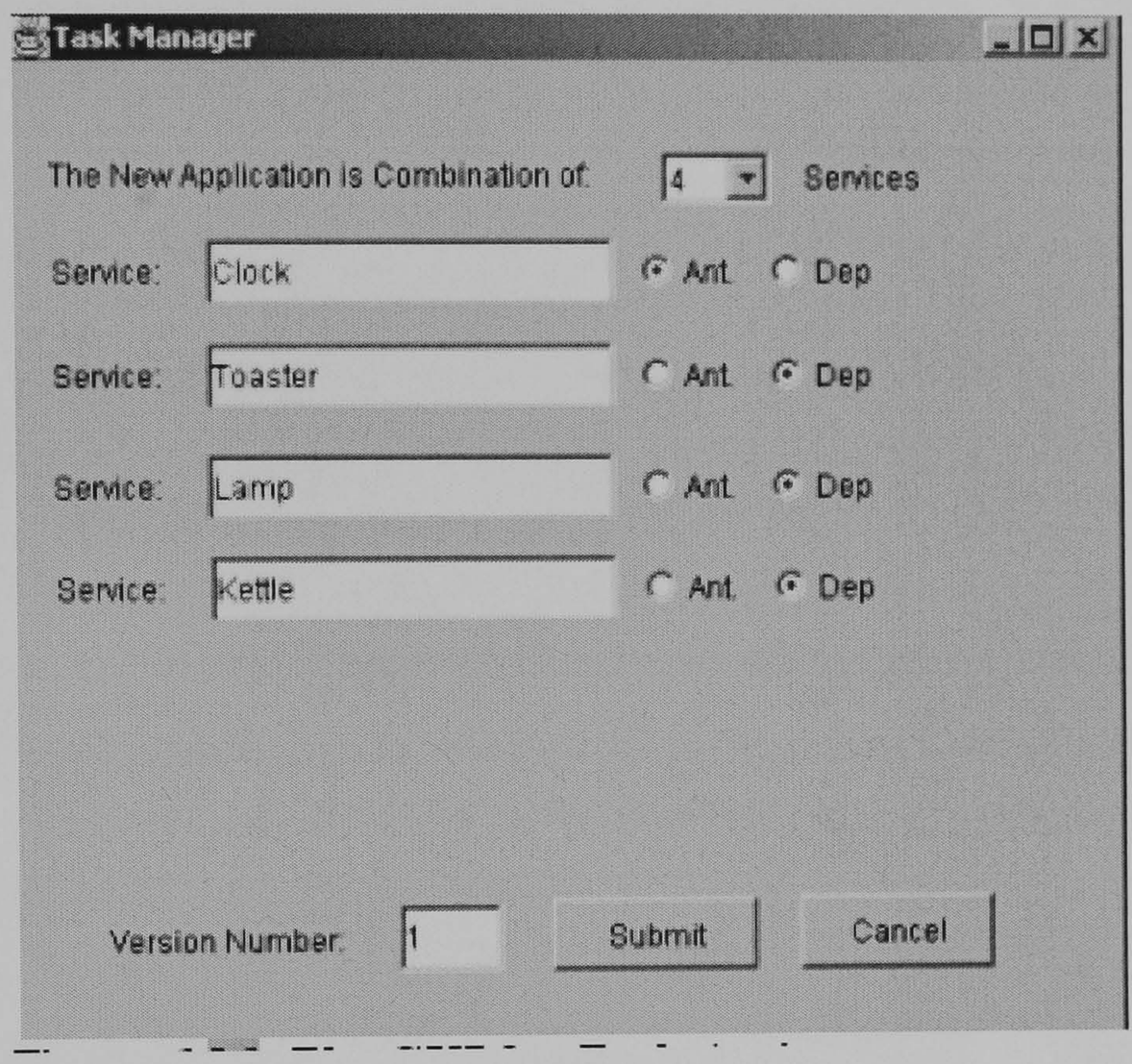



**Figure 6.16: The GUI for Task Assignment**

### 6.6.2 Application 2: Software Services

The second example application was designed and implemented to test the same features of the Impromptu Framework as in the previous example. The difference between these two example applications was in the complexity of the services as networked appliances are considered as simple services. For a software services example we have chosen services such as a Calculator, Dialog Window and Text Editor.

Figure 6.17 shows a screenshot of the GUI of an application that was developed to test the relatively complex software services scenario and concentrating on the invocation of this type of service. The screenshot illustrates a similar GUI to that used

in home the appliances scenario, but in this case it supports the runtime discovery of published software and an assembly utility to compose a new application.

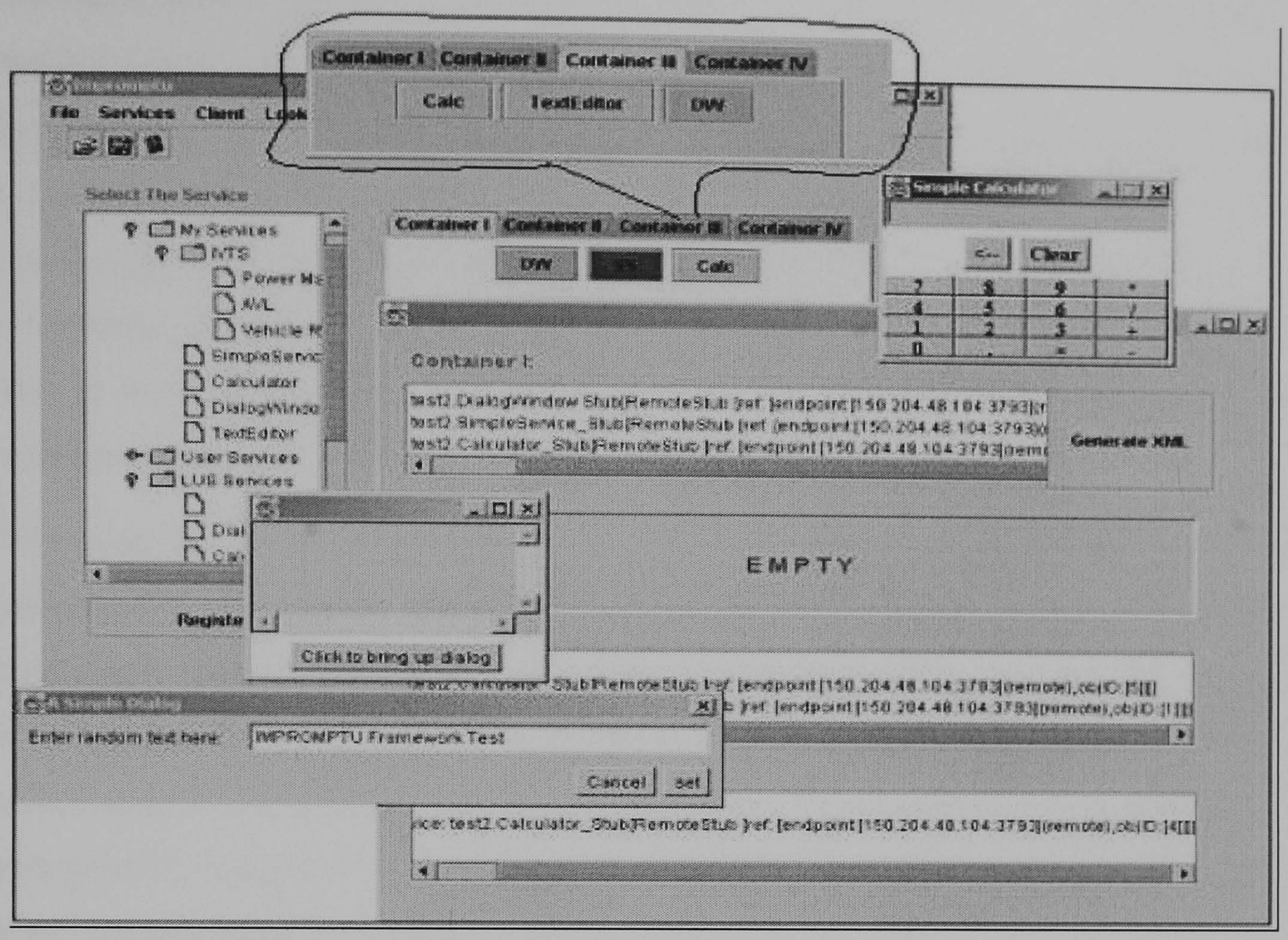



**Figure 6.17: The GUI for software services application**

The service tree here also presents the user with a list of all the services that are available on a lookup service at a particular instant in time. The user can choose the service that s/he wants to use, but in order to invoke the service, the client-side (in this case the invocation sub-service of assembly service) needs to know the method signatures. Unfortunately these are only known to the service developer who is responsible for implementing the service. To address this dilemma, we use XML-based service descriptions that specify method signatures as mentioned previously.

To explain the use of XML and the difference between the scenario described in a previous section and this, we reconsider the calculator that may specify *add, subtract, divide* and *multiply* methods. Although similar to the home appliances application, the calculator can have a single method `getCalcInterface` that displays the calculator GUI by returning a Java `JFrame` object. For this particular application, we consider both cases. One of the novel aspects of our approach, as described before, is that of the XML meta-representation, which is attached to a service as an attribute and passed to the client component via a lookup service. Moreover, this document

contains not only a description of single methods of the services, but also the description of the location of the service interface file. For instance, in case of the calculator, instead of describing the methods the calculator provides (*add, subtract, multiply, divide*), when the XML document is parsed by the client side, the location of the service the interface can be obtained from InterfaceLocation tag (`<InterfaceLocation>`**`Http://cmsegris:8080/userdoc/calcInterface`**`</InterfaceLocation>`). An assembly service, namely the invocation sub-service, receives information about the location of the actual class file using Java's reflection API [67]. This is used to acquire the structural contents of a Java class including the list of methods implemented by the class.

When one component requires a service provided by another component to deliver its prescribed functionalities then there is a dependency relationship between the two components [92]. Conceptually, a dependency is a directed relationship between a set of components, which can be represented using the ideas of [93]who define the relationship based on two roles: the dependent component (a client) and the free or independent component(s) (service provider(s)). Using this representation, we are able to maintain the dependencies that exist for each service within a virtual container (Sec. 6.6.1 and Fig. 6.13). Dependencies may also be dynamic, changing in accordance with different user tasks and these dependencies are represented within the XML descriptions associated with each application service as shown in Figure 6.9 (Sec. 6.5.1).

## 6.7 Summary

This chapter is divided into two main sections. The first section presents the prototypical system implementation called *Impromptu*. The second is the implementation of the OSAD model. Therefore the architecture of this application is prototypical of the OSAD model containing three main services and consequential sub-services.

The Assembly service was described with its sub-services using a UML class diagram. We explained how the on-demand service delivery and assembly could be implemented using specific programming language and middleware technology. We

also explained why Java was chosen for the implementation of the OSAD model and how Jini middleware technology supported development of this application.

The second section contains the description of example applications that were developed to test and evaluate various features and functionalities of the *Impromptu* prototypical system. We have described two example applications namely: Home Appliances and Software Services. We categorised the home appliances as a simple service and the software services as relatively complex.

# Chapter 7

# On-Demand Service Delivery and Self-Healing Software Pattern Language

## 7.1 Introduction

This chapter describes the on-demand service assembly and delivery model as a pattern language with the goal of making the model accessible to software designers. The pattern language we introduce can be used for distributed application development with self-awareness and self-healing capabilities.

The pattern language template is based on Alexander's original pattern language template [94], which has the benefit of generality and independence from any implementation choice including; domain details, context and programming language [77]. The Gamma design pattern [77] style would require tying the patterns to a particular programming language, which was not the purpose of this work.

Alexander's original pattern format is as follows (Fig. 7.1): each pattern starts with a picture to provide an architectural impression of the pattern. The picture has a description. The next few sentences provide the context of the pattern and relate it to the larger patterns in the language. Three diamonds mark the beginning of a section. The problem is stated and highlighted in bold text; and then specific examples and descriptions of the problem are given. The purpose of the problem statement is to show what it would be without having the solution for this problem. One common example will be used for every pattern. Then the solution is stated in bold text as an instruction. A diagram is given to summarize the main idea of the pattern. Three diamonds mark the end of the body section and a paragraph relates this pattern to the other "lower" patterns in the language that are needed to complete it.

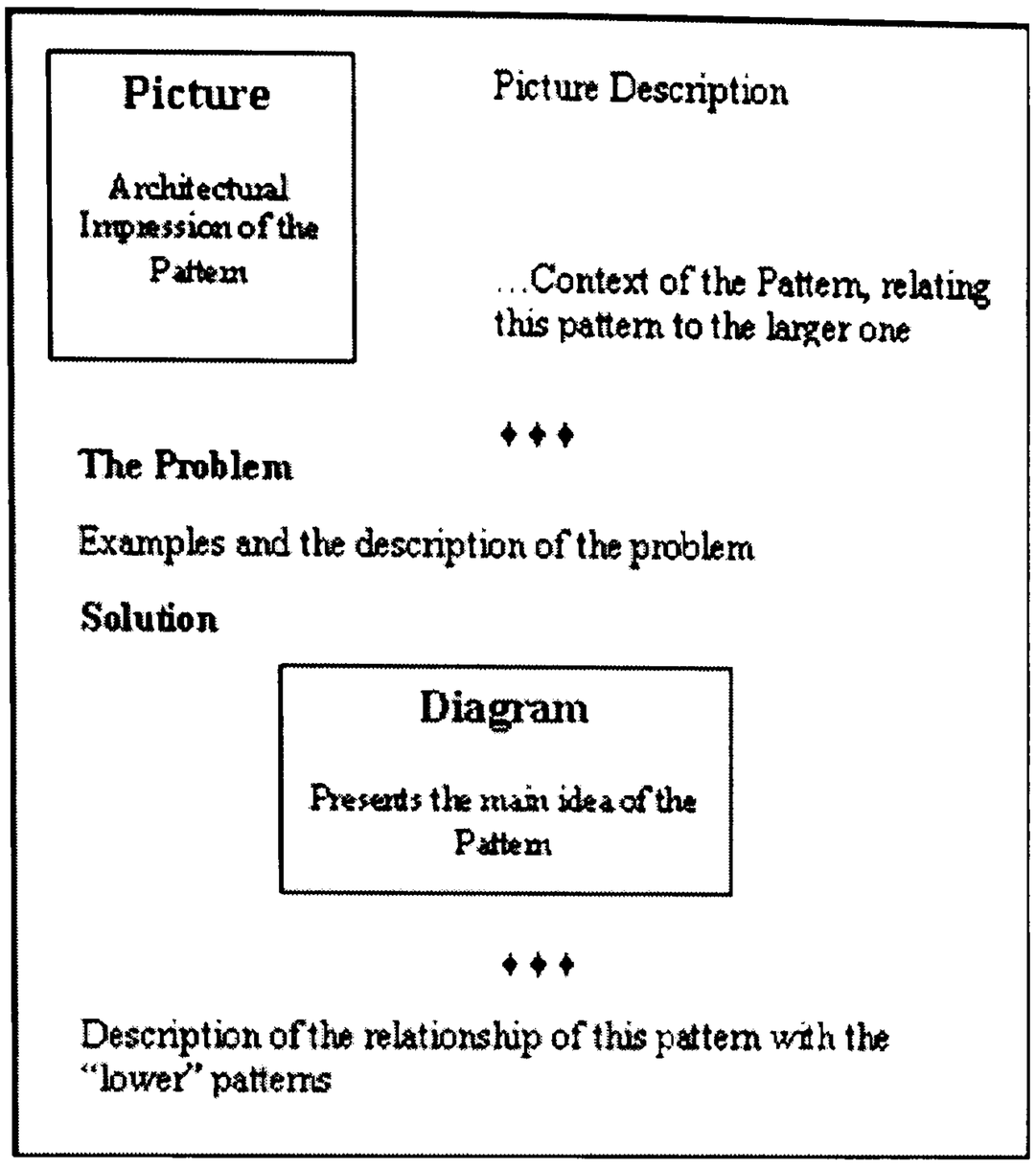



**Figure 7.1: An illustrative example of Alexander's Pattern**

Our pattern language presented in this chapter was implemented in Java and Jini (distributed object and service oriented programming) and tested with different application scenarios, and described in Chapters 6 and 7.

## 7.1 Viable System Model - to Model Autonomic Systems

The proposed On-demand Assembly and Self-Healing software pattern language including the design of the Impromptu Framework have been influenced by Beer's Viable System Model [95] [96] – a general cybernetic management model. Thus it is important to give a brief overview of the model before describing our pattern language.

### 7.1.1 VSM model: a Brief Overview

Beer's Viable System Model identifies the five systems that must exist for any entity to survive in a changing environment. It appears to offer a conceptual blueprint for the design of self-healing, autonomic software. The model explicitly incorporates dynamic planning and self awareness systems, while the classical cybernetics that underpin the model are closely related to control systems theory, so in some respects, the approach lends conceptual support to the guiding metaphors of current efforts in

self-adaptive, autonomic software. A detailed description of the Viable system model and its relationship with self-healing, autonomic software is described in next section. That will be followed with the description of each new pattern in relationship to the Viable System Model.

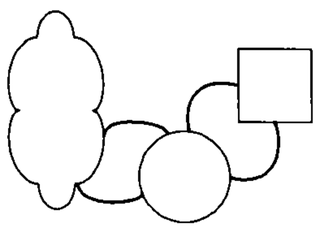
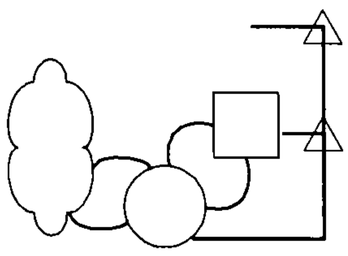
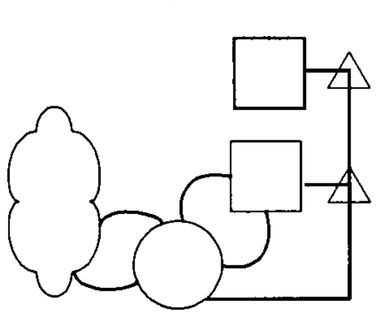
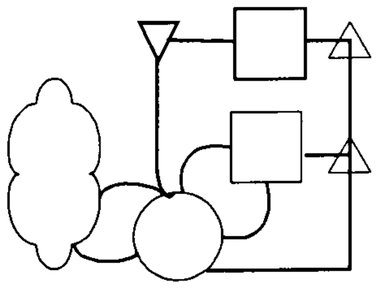
Beer describes the management of knowledge as follow [60]:

> *"You must learn to manage yourself and your formal and informal exchanges and interactions with others. This must be done in the context of your understanding of who you are: your goals, your capabilities, your knowledge of your own strengths and weaknesses; and your appreciation of your social, technical and business environments. Individuals must be able to engage in activities in different 'markets', keep them from interfering with each other, manage them together, focus an eye on the future, and assess their different aspects from the perspective of the 'big picture' of their whole life's narrative."*

Although Beer was talking about individuals, computing systems also have the challenge of maintaining continuity and identity over time – sometimes with minimal infrastructure. They too must integrate and manage their knowledge and information and their exchanges with their environments to perform effectively. The VSM is a powerful descriptive and diagnostic tool to map management capacities to promote viability.

The model identifies the necessary and sufficient communication and control systems that must exist for any organization to remain viable in a changing environment. In doing so, the model does not attempt to specify the activities that must occur in each system, instead activities are typified by cybernetic rational [95] [2] to allow either the designer of activities to match the cybernetic criteria or for actual activities to be identified by their system type and hence assigned to the appropriate element of the model. Such a generalized approach allows the model to be applied to any organization regardless of size.

The six major systems advocated by the model are detailed in Table 7.1 below:

| System Identifier | System Type |
|---|---|
| System One (S1) – Operations  | System One performs the productive operations of the organization. An organization may be composed of a number System Ones, each providing a distinct product or service. Each S1 consists of an operational element controlled by a management process and in contact with the operational environment and is in some respects is similar to the plant/management arrangement adopted by control system theory. |
| System Two (S2) – Coordination  | System Two is concerned with coordinating the activities of S1 units. It is essentially anti-oscillatory in that it attempts to contain or minimize inter-S1 fluctuations. This is achieved by the provision of stabilizing, coordinating facilities such as scheduling and standardization information that is disseminated over all System Ones, but tailored locally to suit individual S1 needs. |
| System Three (3) – Control  | System Three is concerned with the provision of cohesion and synergy to a set of System One units. The management processes contained within this system will be concerned with short term, immediate management issues, such as resource provision and strategic plan production, although strategic in this context refers to planning with existing resources rather than the normally accepted sense. |
| System Three* (S3*) – Audit  | System Three * provides facilities for the intermittent audit of System One progress and provides direct access to the physical operations of the particular S1 allowing immediate corroboration of that progress. This essentially provides additional data over and above that provided by normal reporting procedures. |

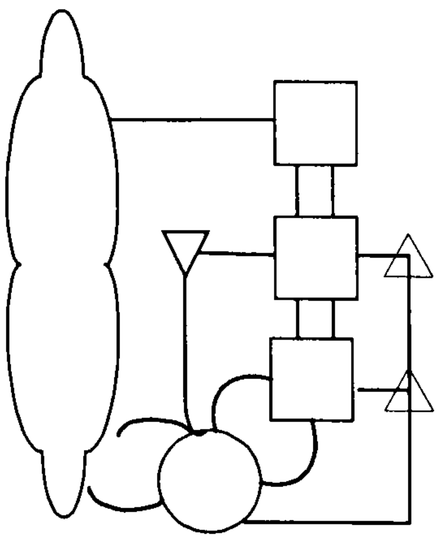
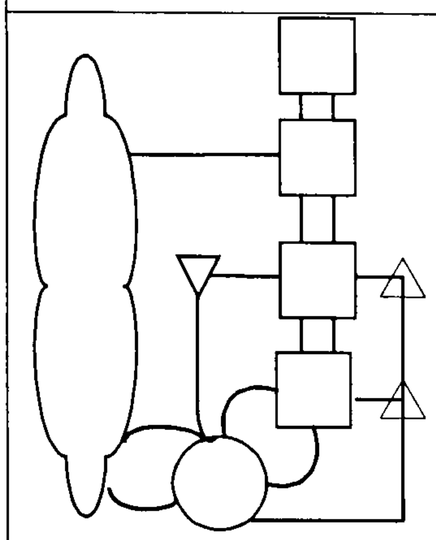
| | |
|---|---|
| System Four (S4) – Intelligence  | System Four is concerned with planning the way ahead in the light of external environmental changes and internal organizational capabilities. S4 'scans' the environment for trends that may be either beneficial or detrimental to the organization and constructs developmental organizational plans accordingly. To ensure that such plans are grounded in an accurate appreciation of the current organization, the intelligence function contains an up-to-date model of organizational capability. |
|  System Five (S5) – Policy | System Five determines the overall purpose of the organization i.e. defines the activities that are performed by S1 as such S5 represents the policy-formulation or normative planning function. Policy formulation is informed by a "world-view" provided by S4 and models of current organizational capability populated by data flowing from the lower level systems in the organization. |

**Table 7.1: The major systems of Viable Systems Model**

The major systems (S1, S3, S4 and S5) are structured hierarchically and connected by a central 'spine' of communication channels passing from the higher-level systems through each of the S1 management elements. These provide high priority communication facilities to determine resource requirements, accounting for allocated resources, alerts indicating that a particular plan is failing and re-planning is necessary and the provision of the "legal and corporate requirements" or the policies of the organization.

The systems described above concern the management structure a one level of the organization, and consequently specify the communication and control structures that must exist to manage a set of S1 units. However the power of model derives from its recursive nature. Each S1 consisting of an operational element and it's management unit is expected to develop a similar VSM structure, consequently, the structure of the systems is open ended in both directions and may be pursued either upwards to ever

wider encompassing systems or downwards to ever smaller units. However, at each level the same structure of systems would occur although their detail would necessarily differ depending on context. This recursivity allows each level in the organization relative autonomy bounded by the overall purpose of the system as a whole.

As mentioned before, the VSM identifies the necessary communications, control and management required for any type of system to remain viable. However, the concept of autonomic computing is to make systems viable without human intervention, which means that systems should manage themselves. This also means that systems should know how to react to changing environments and failures, how to recover from these failures and heal themselves. Therefore from the autonomic software viewpoint, the VSM should serve as a model or type of pattern language for autonomic software development.

### 7.1.2 Viable Pattern Language

In order to express a Viable System as a pattern language, Herring proposed nine patterns that form the viable pattern language [61]. He classifies these patterns as general patterns: **Separate Control (1), Recursive Compositions (8), and Homeostatic Loop** (9), and behavioral- structural patterns: **Operational Control (2), Regulatory Centre (3), Sporadic Audit (4), Adaptive Control (5), Supervisory Control (6), and Alerts (7).** The motivation behind this work is the need to design, and maintain a broad class of software systems, namely, those that must operate in a dynamic environment and interact with humans. The overall goal of using the VSM is to achieve the quality of software viability. Viability is the quality a system has if it can maintain a separate existence – survive -within its environment.

Implementation of these patterns does guarantee the viability of the system, but to build an autonomic system with self-organizing, self-healing features requires more than that. The next section describes how our pattern language contributes to the development of such systems.

## 7.2 Pattern-Oriented Software Architectures for Concurrent & Distributed Systems

Many pattern languages exist today. Some support the creation of specific types of software, such as pattern languages for networked and concurrent computing [97] and enterprise application architectures [98]. Others focus on a particular problem area of relevance in software, such as handling erroneous user input [99], designing user-friendly interfaces to computer and information systems [100], or to make systems viable as described in a previous section [61]. Yet there are other key areas where software researchers and developers expect future pattern languages will be published. One of the areas where the new wave of the patterns are developed and published is in the area of distributed systems development, such as developing service-oriented architectures for web services [101] and flexible self-service applications [69].

An increasing number of patterns are associated with middleware frameworks for distributed services and flexible distributed applications have been documented during the past few years [102] [103]. The next key step is to document the patterns that will come as an additional layer of the existing middleware frameworks (Fig: 7. 2).

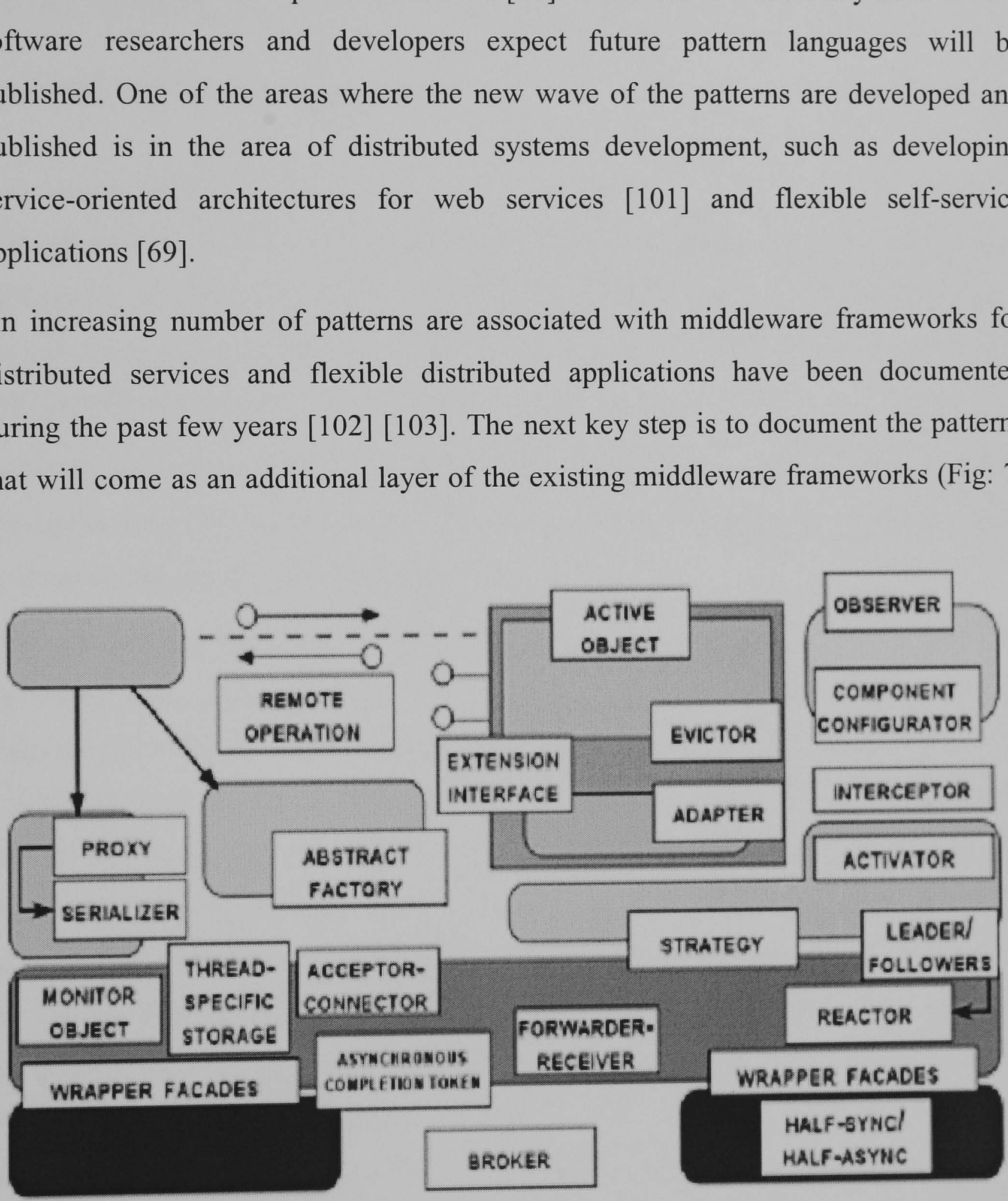



**Figure 7.2: Patterns used in Distributed Middleware Frameworks [102]**

Taking existing work under consideration, the aim of our work was to document such a language that would help software developers to create the additional layers between the middleware and application layers. Some of the patterns described in our work will match with existing patterns for middleware frameworks, and also with the patterns that were documented for viable systems. But we believe that this new pattern language will give general guidelines to autonomic software developers, as it describes all the necessary components for a framework that offers not only viability and self-healing of the system but also the ability to discover and assemble distributed services on demand according to a user's requirements.

### 7.2.1 OSAD Pattern Language

Any computing system that has one, or all of the following features: self-configuring, self-optimising, self-protecting and self-healing can be characterised as an autonomic computing system (Sec. 2.6). Self-healing behaviour is the one that the On-Demand Service Assembly and Delivery (OSAD) model currently implements. Therefore this work contributes to the development of autonomic software.

To do so, the system should have some built-in knowledge or the ability to monitor the operational environment and to learn, to recognise changes and plan how to react to those changes in the environment.

We have used the VSM as a design blueprint for the *Impromptu* framework including the middleware application services, that is, for creating the On-Demand Service Delivery, as evidenced by the proposed Self-Healing software pattern language that will contribute to the development of autonomic computing systems. Viability can be defined as the quality an autonomic system should have to maintain a separate existence and to survive in a changing environment. To achieve this, in *Impromptu* software framework development the software should have the ability to **Lookup (1)** the required services in the surrounding environment. It also should have a function for **Registering (2)** itself and services in this environment and **Discovering (2)** those services that are offered and registered in the lookup service at the current time. The system should know how to **Invoke (3)** the services that were discovered. All the events and functions should be monitored and managed by a **Control and Monitoring Service (4)**. In addition, to make the system autonomic means implementing it with **Self-Healing (5)** behaviour and consequently the system should

have some intelligence. Intelligence can be obtained from the Meta Information submitted by users or generated by the system itself during its life cycle. Finally, the system should have the **Task Assigned (6)** to perform the way the user wants the system to perform.

All the above listed patterns correspond to the system types that guarantee system viability. Therefore, we merged each of them to a particular system type of the Viable System Model. The figure below shows the pattern language for On-Demand Service Delivery and Self-Healing software based on the VSM. Each pattern with its corresponding system type will be described separately in the following sections.

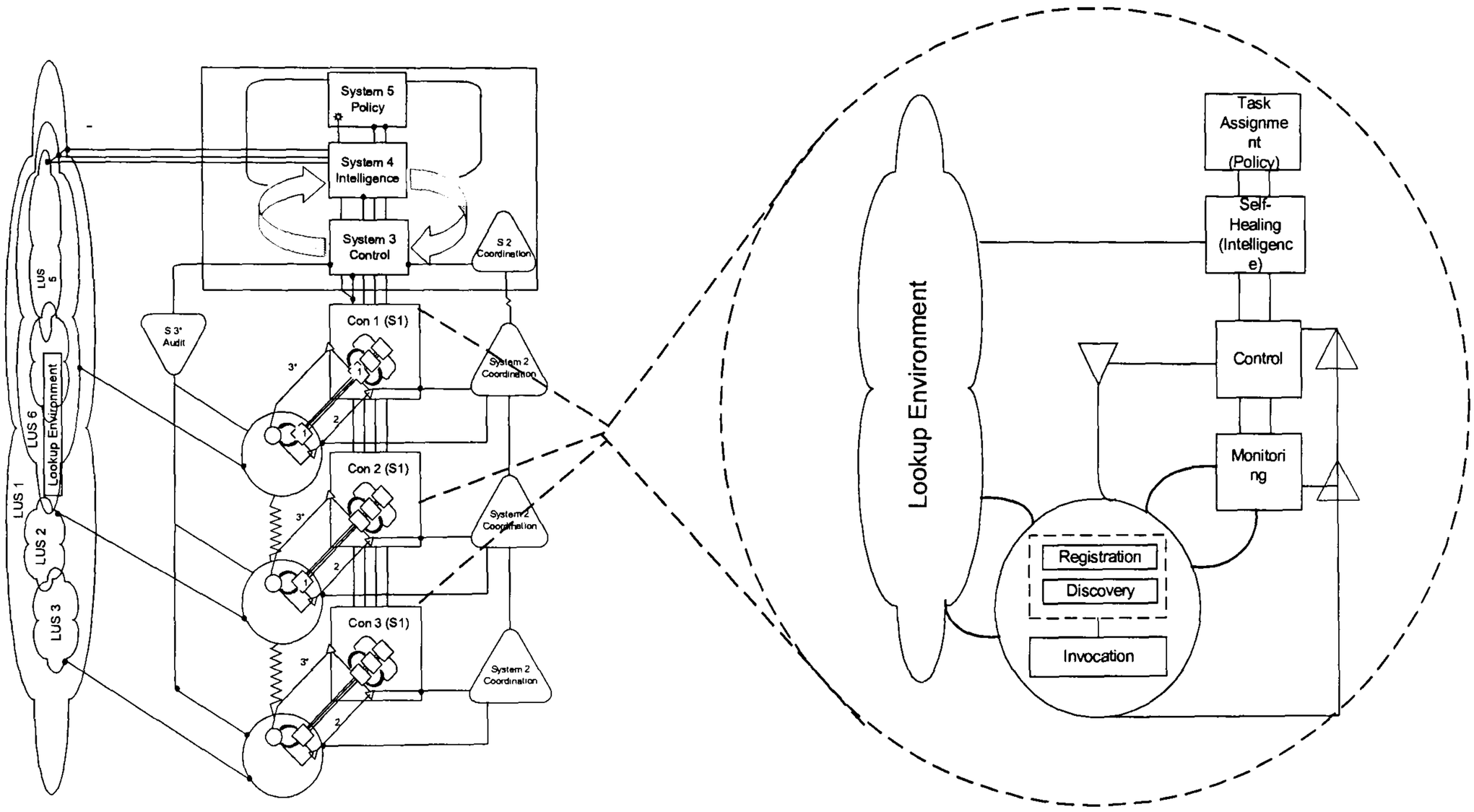



**Figure 7.3: OSAD Pattern Language**

## 7.3 Lookup Service (1)

> *... Any organisation that consists of an operational element controlled by management process is in contact with the Operational Environment.*

♦ ♦ ♦

**The service provider advertises the service and the client requests the service. Therefore, the middleware service or "broker" requires the**

**provider to publish the service and for the client to discover the required service and use it.**

The Problem:

It is now clear that any organization, or in this case, computing system should have an operational element in contact with an operational environment controlled by a management entity to remain viable. In addition, for on-demand and self-healing software, the operational environment requires has access to, a Lookup service. The latter will form a registry service for all published and available software services (be it centralised or decentralised), from which clients can look up (search for required services).

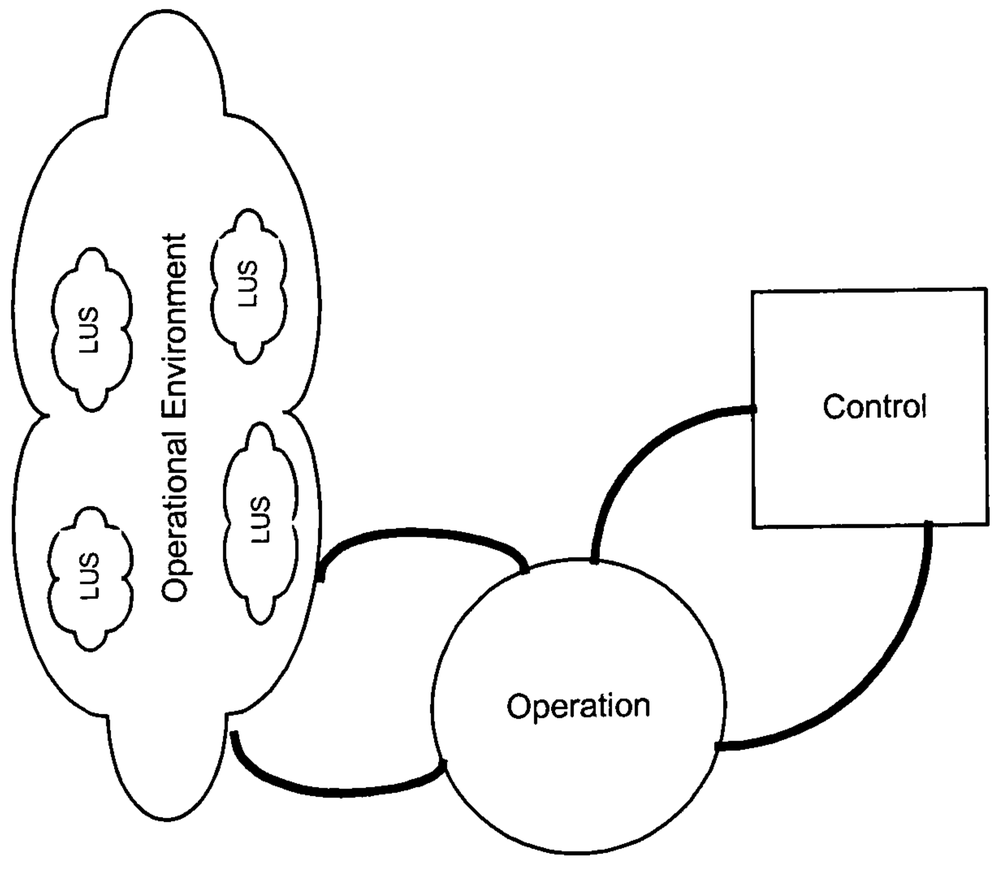
The Solution:

**When building an application or middleware service to support *Impromptu* on-demand discovery and delivery of services, use an existing or develop your own, lookup and broker service.**

Examples:

Good examples of such a Lookup service are the Reggie service found in Jini middleware [22], and the Naming Service from CORBA services [16].

The diagram shown below illustrates the Viable system one (S1), which consists of an operational element controlled by a management process and in contact with an operational environment that consist of a number of Lookup Services.



♦ ♦ ♦

A **Registration (2)** and **Discovery (2)** services are necessary to extend the Lookup Service.

## 7.4 Registry and Discovery (2)

> ... ***Lookup Service*** *(1) enables service providers to register the service and for the client to discover the desired one.*

♦ ♦ ♦

> **All published services should be registered with the lookup service and clients should be able to discover services by searching the look up service.**

The Problem:

Having the Lookup Service in place, service providers need to implement method calls to locate a lookup service, both to publish (register) their services and to discover other registered services from the lookup service. As such, assume that a service provider can act as a server and a client.

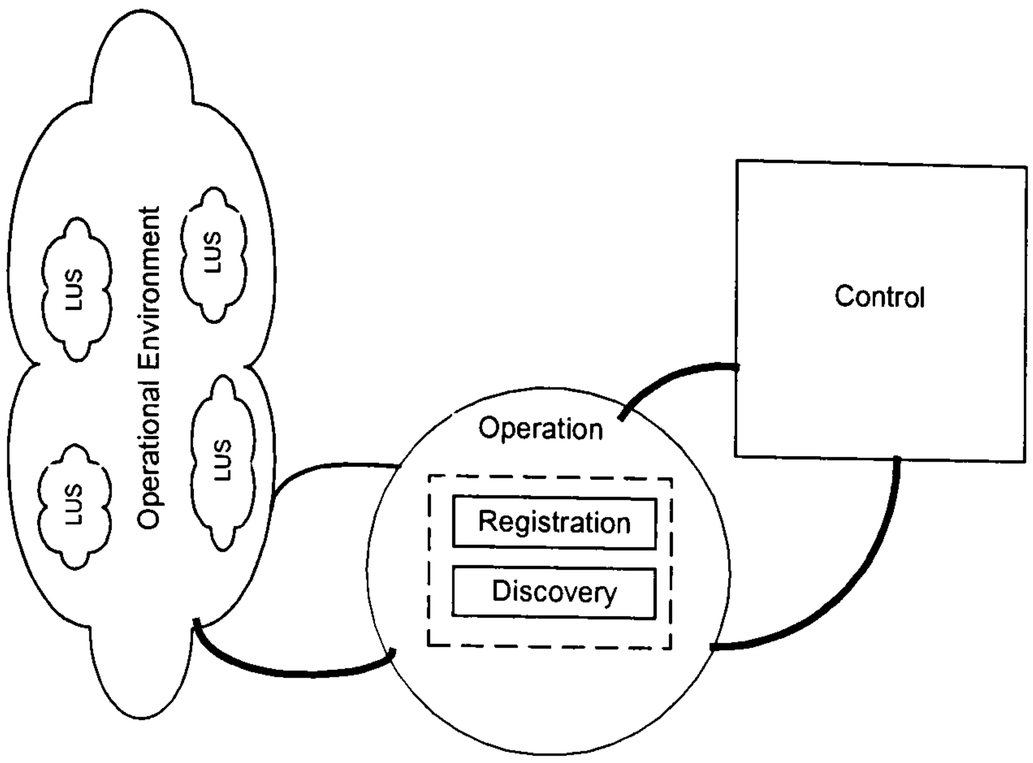
The Solution:

> **When developing application services and distributed applications they must implement methods calls for discovery, service registry and service discovery.**

Example:

Examples can be found and supported by both Jini and CORBA middleware.

The diagram below shows the System One of the VSM. The Registration and Discovery services are presented as operational elements of this system type.




♦ ♦ ♦

After the Services are discovered, the **Service Invocation (3)** methods are required to make of use of the services.

## 7.5 Service Invocation (3)

*... Having Discovered (2) the required services, the execution of the service should take place.*

♦ ♦ ♦

**Services should be invoked in order to make use of them.**

The Problem:

After the service is discovered, the main problem the system will face is the invocation of that service. Although many existing technologies offer this type of service, it is still difficult to define a generic pattern for Service Invocation for a cross standard invocation protocol. For instance, for clients to invoke a given service they need to know the invocation method name and attributes including location. Thus for ubiquitous and automated on-demand service invocation (see self-healing) a cross-standard invocation method is required. The most generic way to describe the implementation of this method will be to request the service provider to register the service and provide a meta-description of the service. Such a meta-description

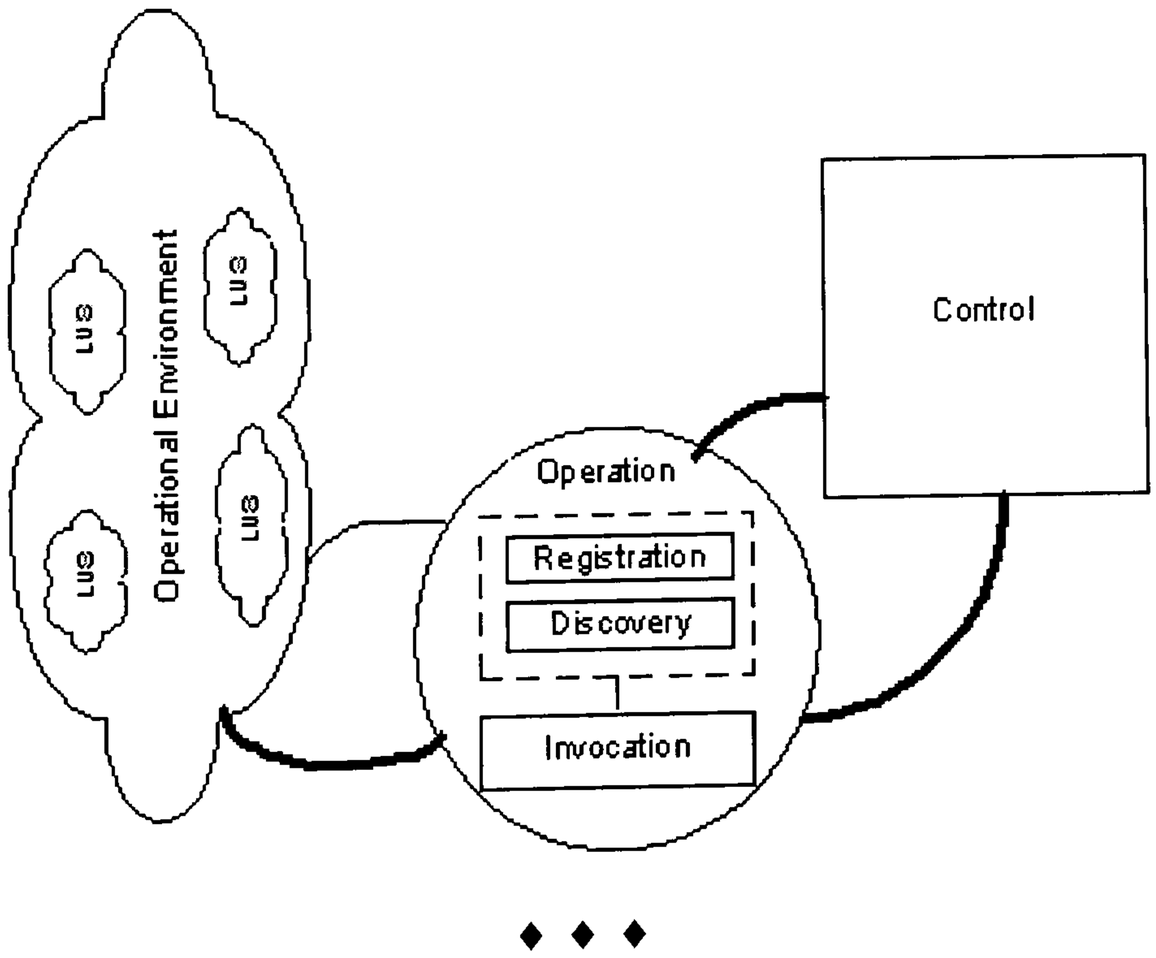
document should be encoded in an open standard language containing for instance; the service signatures, such as: port, host and invocation method. With such an open standard service description can then be automatically used by the Service Invocation component to invoke the service.

The solution:

> **When building the Invocation Service for the On-demand Service Delivery, software needs to implement method calls to access the meta-description document of a considered service, which can be parsed and invocation parameters passed to the invocation method call.**

Examples:

Many technologies support the invocation of service like Java RMI, Jini, UDDI, and Bluetooth SDP, which often provide APIs for the invocation of their native services. For instance, as Jini service invocation is based on the Java RMI protocol, this requires a client to know the method name. This requirement is met if the service is discovered together with its meta-description document encoded in XML. This is similar to the web services protocol. Though, it requires further development to support automated service assembly and generation of invocation code to fully support the on-demand service provision and self-healing [104] [105]. The figure below illustrate a VSM-style representation of the invocation service.




◆ ◆ ◆

As the operational elements of the VSM are in place, the system can maintain its viability and some level of autonomy. Now the software should **Monitor and Control its Service (4)**.

## 7.6 Control/Monitoring Service (4)

> ... ***Service Invocation*** *(5) brings the service to life in an Operational Space. They need to be monitored and controlled.*

♦ ♦ ♦

> **In order to detect runtime service failure, each service or service group must be monitored.**

The problem:

For the system to maintain its efficient and optimal operation, continuous monitoring and control is required. The control service should look after the operations of the process. In biological systems, this is called autonomic regulation and permits living systems to automatically maintain stability, e.g., to breath, to walk, to maintain heart rate. However, the Control/Monitoring Service of the autonomic software system requires more than just autonomic regulation. In biological systems the brain does not need to know how to plan regulations, but in computing systems this knowledge is necessary. If the system has meta-knowledge it can monitor and deliberate on the state and repair of the operational element when required. This will underpin the support of self-organisation, self-healing and/or self-governance.

The Solution:

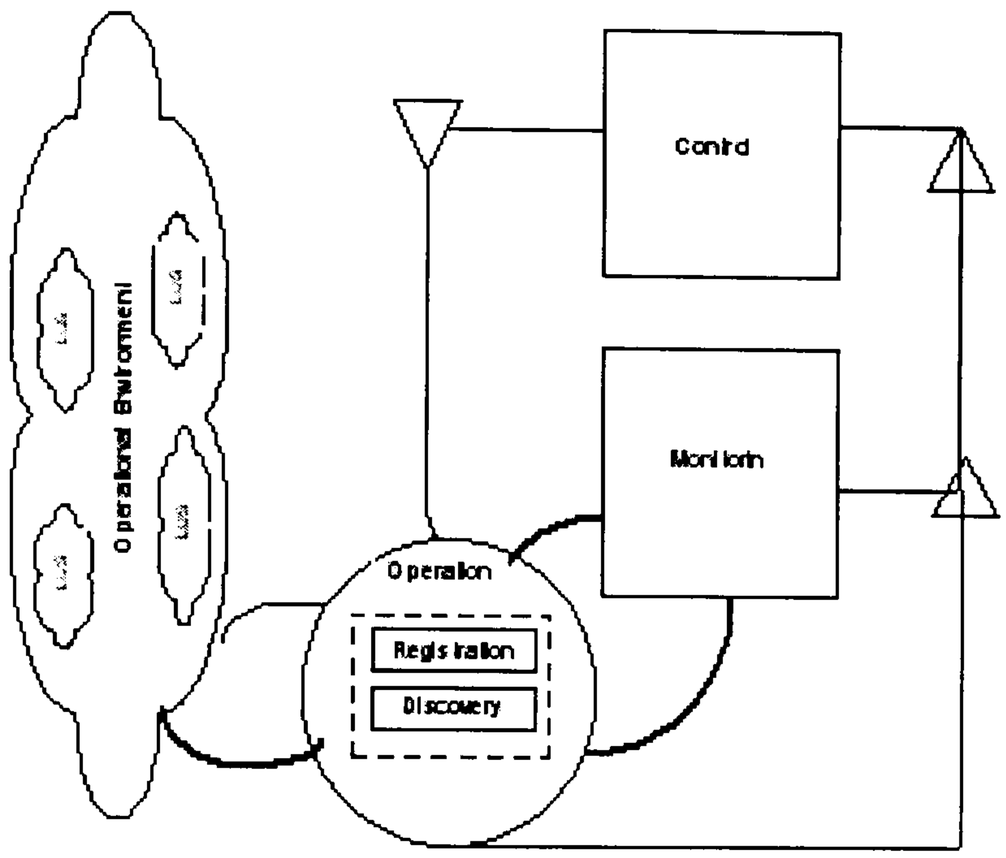
> **Build the software with a control mechanism. Create a process or a set of processes under this control. Provide specific control knowledge to monitor and act in the event of any inconsistency that has been detected, such as failure or performance degradation.**

The control mechanism should be responsible for the management of the overall system including; meta-services and applications. Therefore when designing and building such a mechanism ensure that this mechanism contains a monitoring capability or can access monitoring measurements.

Example:

A good example of a Control mechanism implementation would be the work done by Badr [8]. This work described all a details of building the control mechanism for viable autonomic software.

The following figure shows the S3 of the viable system representing the Control Mechanism for the autonomic software. The Control system is in relationship with the present operational elements and the operational environment.




♦ ♦ ♦

Having the Control Service in place together with its monitoring facility, the system requires a **Self-Healing Mechanism (5)** to provide self-adaptive capabilities.

## 7.7 Self-Healing Mechanism (5)

> *... If the **Control/Monitoring Service (4)** detects any failure of the service, the system should be able to recover from the fault with minimal external intervention.*

♦ ♦ ♦

> **The system should be aware of its structure in order to perform self-healing.**

The problem:

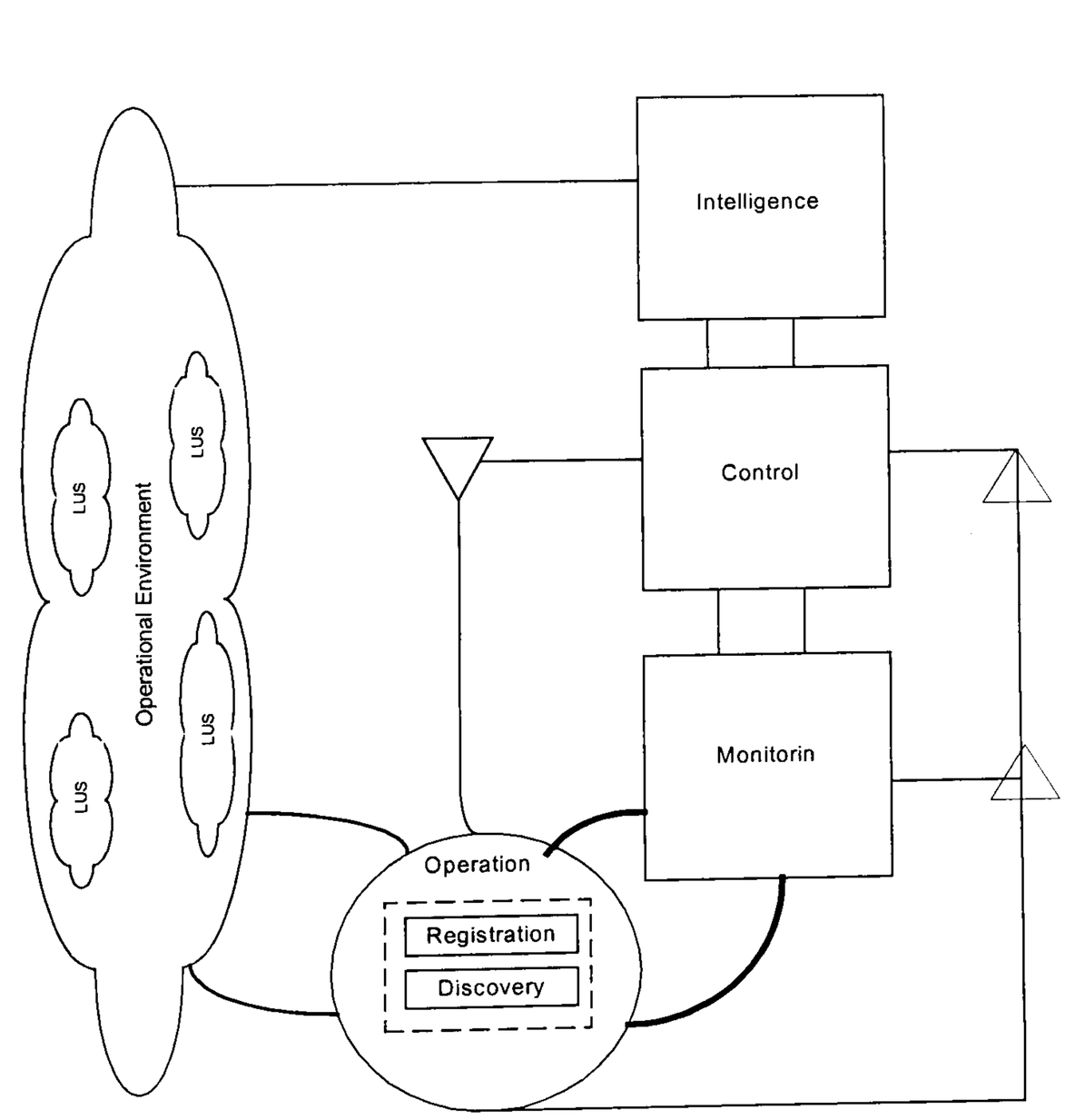
Due to either unexpected changes in the environment or unpredicted systems failure, the system is required to adapt to changes by, for instance: evolving to match the new environmental states, or repair its structural model to re-establish lost functions. Not may software systems can provide such self-healing functions and this is currently performed by human designers and/or systems administrators. But the purpose of the language we are offering is to reduce human interaction and to increase the autonomy of the software by providing self-healing behaviour.

The self-healing mechanism should provide the ability for the software "to scan" the environment for trends that may be either beneficial or detrimental to the organisation and construct developmental organisational plans accordingly.

The Solution:

> **The Self-healing behaviour is implemented by scanning the environment and the systems behaviour and triggering repair action when required.**

Building a self-healing, adaptive controller has been the subject of some recent research work such as: Badr [8] and Herring [61]. The figure below demonstrates our view of the self-healing mechanism fitted as S4 of VSM.




♦ ♦ ♦

Task **Assignment (6)** serves as a kind of policy-driven job allocation service.

## 7.8 Task Assignment (6)

> *... Self-Healing Controller (4, 5) can not make the system autonomic or viable if the system is not aware of the task to perform.*
>
> ♦ ♦ ♦
>
> **Even an autonomic system will not start functioning without task allocation and execution. The system should have access to the users' requirements model.**

The problem:

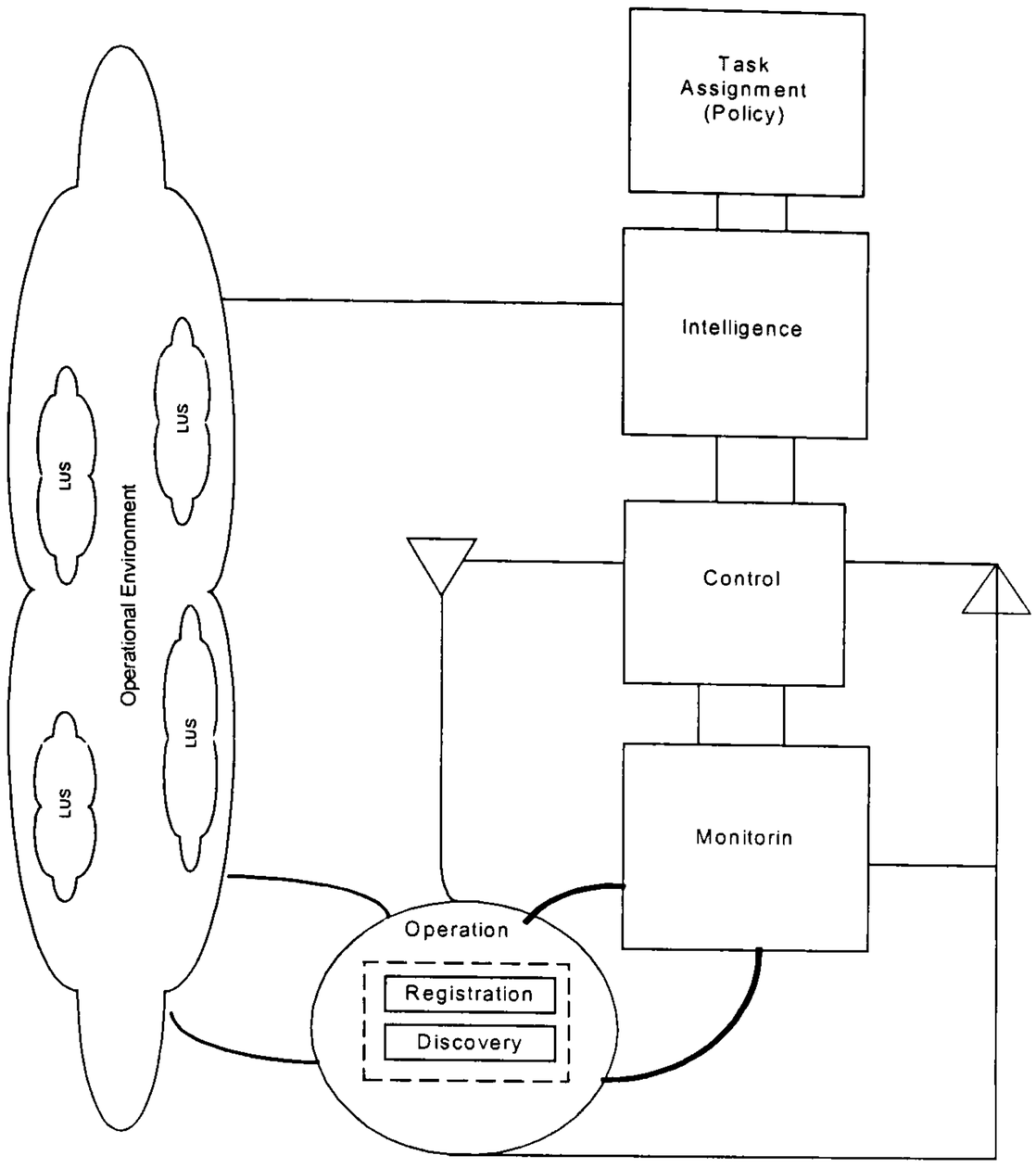
The problem is to be able to set high-level goals to solve or undertake a specific act. This is often a difficult task often performed by human users. Thus, human users must set the goals and directions of the system. Another problem is how to make the system aware of human needs.

The Solution:

> **Design a mechanism, language or user interface to enable the dialogue and/or input of the systems policies including permissions and authorisation and the specification of goals and tasks.**

At the highest level, task assignment is the domain of humans and is accomplished via a human-computer interface. It can be implemented via an appropriate GUI including multi-modal user interfaces (eg speech and sentient interfaces).

The diagram below represents the S5 of the VSM that from our point of view matches the Task Assignment pattern.




♦ ♦ ♦

This pattern completes the sequence of the viable system model. Therefore it completes the sequence of the On-Demand Service Delivery and Self-Healing Software Pattern language too.

## 7.8 Summary

This chapter described six patterns that form a pattern language for autonomic software development. In addition to this, the Viable System Model was used as a model to describe the pattern language, as it offers a conceptual blueprint for the design of self-healing, autonomic software.

Each pattern was described separately using the Alexander pattern description style, to give software developers a wider choice in the technologies and programming languages to use in systems development. The widely used Gamma design pattern style would require tying the patterns to a particular programming language, which was not the intended purpose of this work.

# Chapter 8

# Evaluation

## 8.1 Introduction

This chapter presents an evaluation of the on-demand self-healing software framework described in this thesis. The evaluation of such a model and associated software is not an easy task, as there is no straightforward way of evaluating distributed applications with ad-hoc self-healing capabilities, nor are there any clear metrics or accepted benchmarks.

The remainder of the chapter outlines the evaluation methodology, before undertaking a quantitative and qualitative evaluation of the work. Finally, the chapter concludes with a critical analysis and general discussion of the results.

## 8.2 Methodology

The evaluation has been designed to demonstrate the use, and effect of the implemented self-healing capability of the On-Demand Service Assembly and Delivery (OSAD) model from both qualitative and quantitative perspectives. In other words, we will analyse the effect of the self-healing behaviour on the prototype software performance overhead, which is, for convenience restricted to only processing time.

### 8.2.1 Objectives

For the purpose of the evaluation, we have developed two test example applications using the OSAD framework, namely: sorting algorithm services and home appliances. In each case we compare the prototype software system performance profile with and without self-healing behaviour. For instance, we use elapsed time taken to sort a given array and we measure the applications performance profile with and without the self-healing behaviour [106]. The "performance profile" is here used mainly to indicate singularity points, for instance, when the autonomic behaviour has taken place. Hence, the evaluation is not designed to perform a performance analysis or a

comparative analysis of the chosen sorting algorithms. To this end, and in line with the OSAD model, we developed each of the chosen sorting algorithms as a separate OSAD services that will be discovered and executed at runtime, including after failure has been detected – the triggering of the self-healing process. Finally, we describe the evaluation from a qualitative perspective too. This evaluation will use a set of metrics including:

Stability: this is one of the most important metrics for software with self-controlling behaviour. The system is stable if its responsiveness to control rules is in a desirable interval. In our case, the system is stable if its responsiveness to the service failure and self-healing time is in an allowable time range.

Time Performance Profile: this measures the amount of time taken by the system to complete the whole process, and in which the value of the self-healing time is within the desirable range.

Average Latency: this measures the average time required for the system to start up the self-healing process.

### 8.2.2 Approach

Although this evaluation is not intended to be a formal performance evaluation of our OSAD model and its associated software, nevertheless, we use elapsed time as a performance profile metric to outline the effect and overheads of *ad-hoc* self-healing capabilities on systems' performance.

For calibration purposes, prior to the evaluation, 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 applicable, for instance, to the sorting algorithm services.
- Defining upper and lower performance limits for given applications, which will provide some type of knowledge, for instance, to guide the system to perform self-healing to maintain a specified overall system performance.

- Measure the self-healing latency time, which is used as a nominal time tolerance measure.

### 8.2.3 Overall Settings

As described in Chapter 6, the prototype applications used for this evaluation were designed in line with the OSAD model and extending Jini middleware. So, all of the OSAD services were developed as an additional layer between the Jini middleware and the applications layer. The *Assembly Service* and *Operational Service* of the OSAD model guarantees the service on-demand delivery and assembly, but the *System Manager* is responsible for triggering or initiating self-healing following system failure detection (Fig. 8.1). A detailed description of the design and implementation of the OSAD model is provided in Chapters 5 and 6.

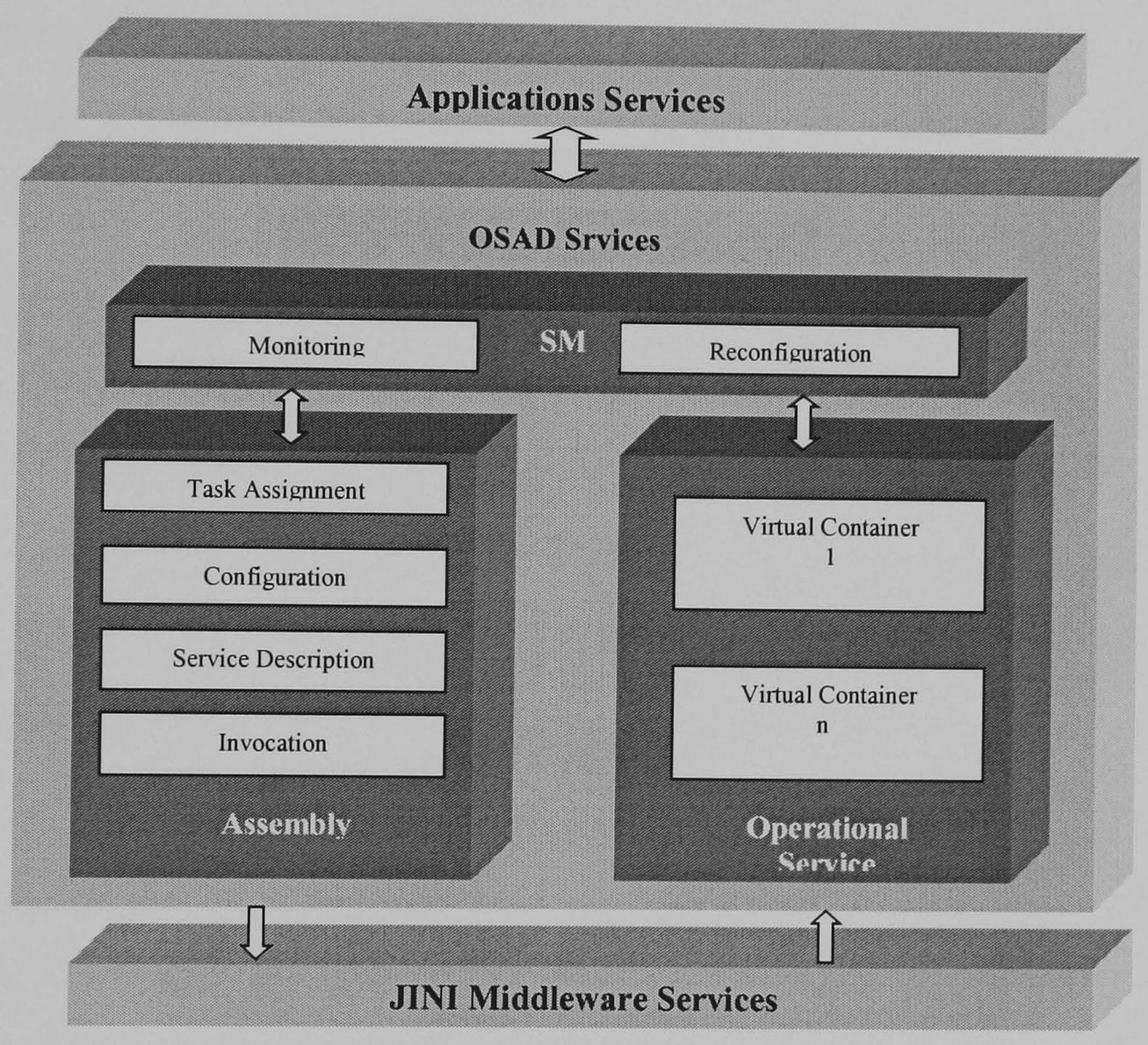



**Figure 8.1: The architecture of the OSAD model**

#### 8.2.3.1 Example Applications

For this evaluation, two different user applications have been developed, which are sorting algorithms and the home appliances application. In particular, the sorting algorithms scenario is based on the use of three main sorting algorithms, namely: Bubble, Quick and Insertion sorting algorithms [107]. The home appliances and the software services application are here used specifically to evaluate the performance of

the system when it is under high load. Each of the example applications is evaluated using the following scenario:

- The user submits a task, which is translated into an XML document by the system.
- This document is parsed and the information is passed to the Discovery service, which is one of the sub-services of assembly service. Discovery of the required services is done by the Discovery service.
- It is assumed that distributed services are registered and discovered with their additional description files written also in XML. The invocation service can parse this document thereby providing meta-information about the service and obtain the invocation method. Then the required service is invocated using Java RMI. At the same time the service is put into one of the virtual containers of the Operational service. This procedure will be performed for each required service.
- A new XML document will be generated by the Assembly service for each Virtual Container and describes the assembly structural model (composition, and the list of all the member services (component services)).
- The Monitoring service monitors each virtual container and in the case of a detected failure in one of the services, sends a failure notification event to the Service Manager using the Java remote events model.
- The Service manager triggers a recovery process via the discovery and binding of an "equivalent" to replace the failed service. The discovery and invocation are performed by the assembly service as before. Then the service manager performs reconfiguration of the new application.

The sequence of actions carried out for both of the example applications are as follows:

- For each task a user submits for execution, the Assembly service provides the following services:

  *Discovery* – using the Jini `LookuDiscovery` method, with the service name parameter.

*Invocation* – by parsing the meta-information document that comes with the service. In a simple service case just by obtaining the invocation method, or in complicated cases by using the Java Reflection API [67] in order to obtain the invocation method. Finally, the invocation is performed using Java RMI [108].

*Configuration* – by placing the services from the same group on one of the Virtual Containers.

- Each Virtual Container is monitored and in case of the failure of one of the services the remote event [6] with the name of the failed service is sent to the System Manager to start the recovery process.
- When the remote event notification of failure is sent, the system starts the recovery process that evaluates the self-repairing, self-healing process.
- The elapsed time for each recovery process is measured and compared with previous measurements (to ensure that all results are in the acceptable range). The metric unit used to evaluate the self-healing (runtime recovery) process efficiency in OSAD model was time in milliseconds.
- The average latency is also used to monitor the system performance as a whole.

### 8.2.5 Environment

The evaluation is performed using an XP 1800 Athlon (tm) ~ 1.5 GHz processor with 261 MB of memory, and running MS Windows XP operating system and connected via Ethernet. The example applications and the OSAD model were implemented using the Java programming language (JDK 1.4), and Jini 1.1 middleware.

## 8.3 The Quantitative Evaluation

This section provides a detailed description of the Quantitative Evaluation of two different example application scenarios, namely: The Sorting Algorithms and Home Appliances. At the end of each Scenario the evaluation results are presented. Performance of failure detection mechanism is also described in this section.

### 8.3.1 The Sorting Algorithm Services Scenario

Sorting algorithms are generally recognised as an important benchmark in scientific and commercial applications [109]. For this evaluation, we have chosen three well known sorting algorithms namely: Bubble Sort, Quick Sort and Insertion Sort, each of which is developed as individual Jini software services.

In this experiment, we run the application 15 times for each sorting service (algorithm), each time we increased the size of the randomly generated arrays. Then the time for service invocation plus sorting was recorded (Table. 8.1). The experiment is outlined as follows:

- Each sorting algorithm was implemented as an individual Jini service (`BubbleService, QuickService` and `InsertionService`). All services are registered in the lookup service.

```
public void register() {
    BubbleService bs; //(QuickService cs, InsertionService ins)
    LookupLocator lookup;
    Entry[] attributes;
    JoinManager joinmanager;
   try {
     attributes = new Entry[2];
     attributes[0] = new Name("BubbleService");
     attributes[2] = new
Comment("http://cmsegris:8080/userdoc/ServiceDescription.xml");
      bs= new BubbleService();
      joinmanager = new JoinManager
  (
    bs,
    attributes,
    bs,
    new LeaseRenewalManager ()
  );
      }catch (Exception e)
          {
          System.out.println("server: MyServer.main():
Exception " + e);
          }
 }
```

**Figure 8.2: The registration method for sorting services**

- The array size was initialised in the experimental application by calling the `getArrSize()` method (Fig 8.3).

```
public int getArrSize(){

int array_size = 0;
try{
array_size =
integer.parsInt(sortingAlgFrame.jTextField1.getText());

          }
catch(Exception e)
return array_size;

}
```

**Figure 8.3: The initialisation process of the array size**

- The sorting method for each algorithm was implemented as an invocation method for the corresponding Jini service (BubbleSort(), QuickSort(), InsertionSort() (Fig 8.4)). A randomly chosen size of array was passed for sorting and each service was invoked separately.Figure 8.4: Invocation methods for three sorting services

```
// Invocation method for Bubble Sort Algorithm
public void BubbleSort(int a[]){

    for (int i = 0; i<a.length; i++)
  for (int j = a.length-1; j >i; j--)
  {
     if(a[j-1] > a[j])
     { int T = a[j-1];
        a[j-1] = a[j];
        a[j] = T;
        } } }
//Invocation method for Quick Sort Algorithm
public void QuckSort(int a[], int left,  int right){
   . . .
//implementation of the method
      if( left < hi )
            QuckSort( a, left, hi );

         if( lo < right )
            QuickSort( a, lo, right );
   }   }
private void swap(int a[], int i, int j)
   {
      int T;
      T = a[i];
      a[i] = a[j];
      a[j] = T;
   }
// Invocation method for Insertion Sort Algorithm
public void InsertionSort(int a[]){
for (int i = 1; i < a.length; i++ )
        {
            int value = a[i];
            int j;
            for ( j = i - 1; j >= 0 && a[j] > value; j-- )
             {
                a[j+1] = a[j];
                        }
         a[j+1] = value;
} }
```

- The time in milliseconds for the service invocation and sorting was measured in each case. The elapsed time was calculated by the difference between the initial time and final time measured by the `System.currentTimeMillis()` method of the service invocation and sorting process respectively. Figure 8.5 shows an example calculation of the elapsed time of the `BubbleSort` service. The elapsed time for other services is calculated the same way.

```
long startTime = System.currentTimeMillis();
     classifier.BubbleSort(array_size);
long endTime = System.currentTimeMillis();
int elapsedTime = (int)(endTime-startTime);
```

**Figure 8.5: An example of calculating the elapsed time for the service invocation and the sorting algorithm**

- The calculated elapsed time is used to identify the boundary range and algorithm intervals. This is necessary to show the efficiency of those three services working together and to generate the control rules for the control mechanism [8]. For each algorithm, the measured elapsed time indicates 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 one service reaches the limitation another service is required to be invoked. For the purpose of the experiment and evaluation it is assumed that the limitation of one service causes a failure. So, in order for the system to continue functioning the other service should be found and invoked.

- At this point, the *Service Manager* is sent an event notification (using a Jini Remote Event) of which service is required to be invoked. The system does the service discovery through the service name and then invokes this service. This process involves the following steps:

  * Discover the service by name

  * Get the location of the service description XML document

  * Parse the document and get the invocation method

* Pass this method to the Invocation Service

- The latency (elapsed time) for all these processes is calculated using the same method as described in step 4, where we were calculating the elapsed time for each service separately, e.g.:

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

### 8.3.2 The Experimental Results

The experimental results are presented as follows:

- The number of elements being sorted in a given randomly generated array.
- The same size of array took different time to sort by each sorting algorithm.

The results shown in Table 8.1 illustrate that the processing time increases and the system slows down as the array size increases. So, we ran the experiment for each service to find the ideal choice of an array size for each service. During the experiment we have discovered that when the array size is too small no useful information is elicited (such as zero or a very small number). With too large array size the time for the service running is too long. We have picked different sets of array sizes for each different service. These are the cases when the service running time is not zero or too long. The results for all services are shown in the following table:

| Bubble Sort Service | |
|---|---|
| Array (size) | Time (ms) |
| 1500 | 20 |
| 3000 | 31 |
| 4500 | 90 |
| 6000 | 150 |
| 7500 | 200 |
| 9000 | 311 |
| 10500 | 401 |
| 12000 | 541 |
| 13500 | 761 |

| Quick Sort Service | |
|---|---|
| Array (size) | Time (ms) |
| 20000 | 10 |
| 40000 | 20 |
| 60000 | 30 |
| 80000 | 40 |
| 100000 | 50 |
| 120000 | 60 |
| 140000 | 70 |
| 160000 | 81 |
| 180000 | 90 |

| Insertion Sort Service | |
|---|---|
| Array (size) | Time (ms) |
| 200000 | 20 |
| 300000 | 50 |
| 400000 | 70 |
| 500000 | 81 |
| 600000 | 90 |
| 700000 | 100 |
| 800000 | 131 |
| 900000 | 141 |
| 1000000 | 150 |

| | |
|---|---|
| 15000 | 951 |
| 16500 | 1052 |
| 18000 | 1302 |
| 19500 | 1332 |
| 21000 | 1545 |
| 22500 | 1793 |

| | |
|---|---|
| 200000 | 100 |
| 220000 | 151 |
| 240000 | 210 |
| 260000 | 220 |
| 280000 | 260 |
| 300000 | 271 |

| | |
|---|---|
| 1100000 | 260 |
| 1200000 | 280 |
| 1300000 | 300 |
| 1400000 | 210 |
| 1500000 | 250 |
| 1600000 | 331 |

**Table 8.1: The calculated elapsed time for three different services**

The Figures 8.6, 8.7 and 8.8 show a linear scale of 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).

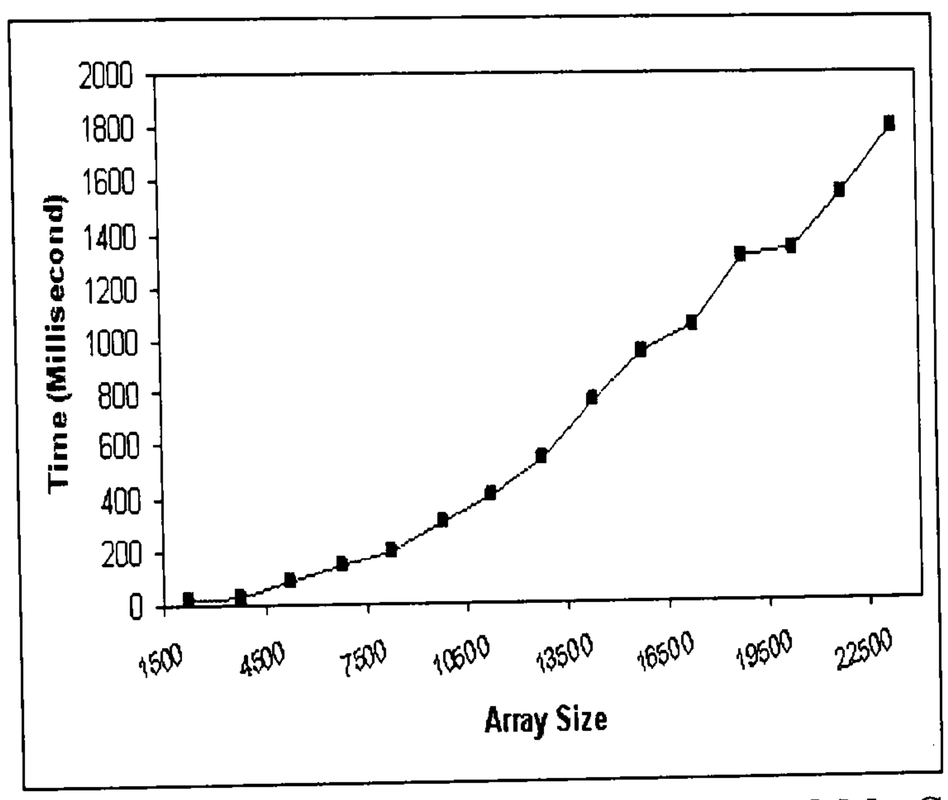



**Figure 8.6: The elapsed time for the Bubble Sort Service**

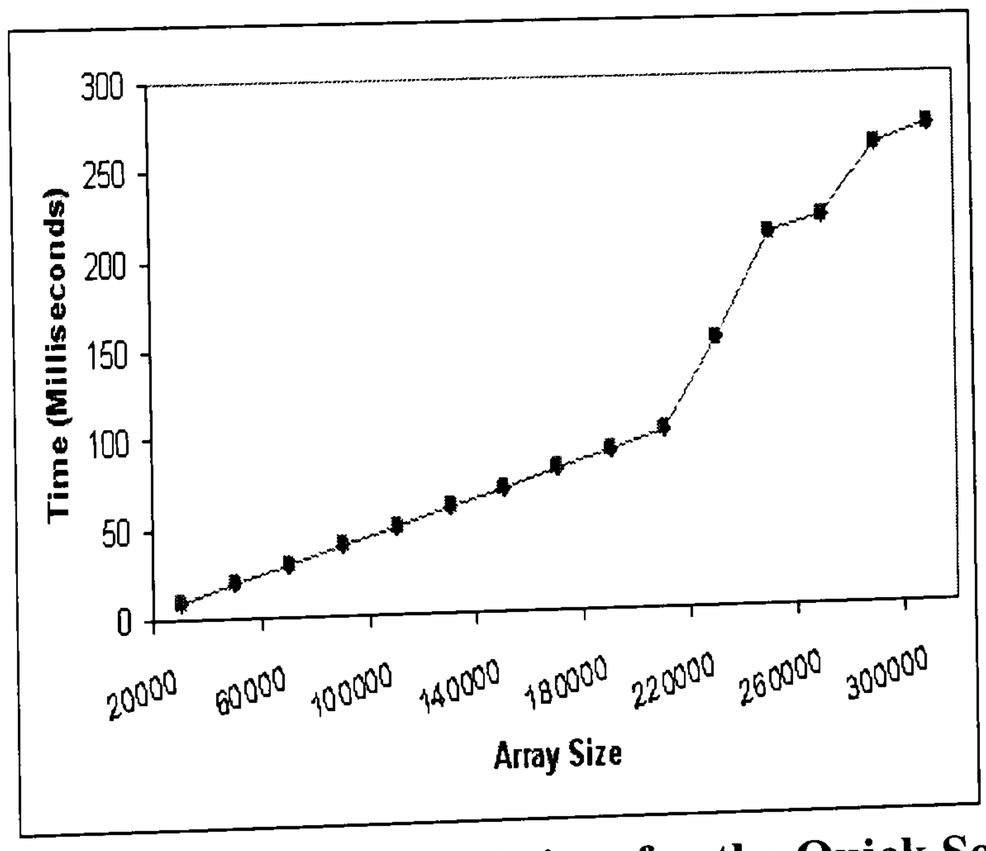



**Figure 8.7: The elapsed time for the Quick Sort Service**

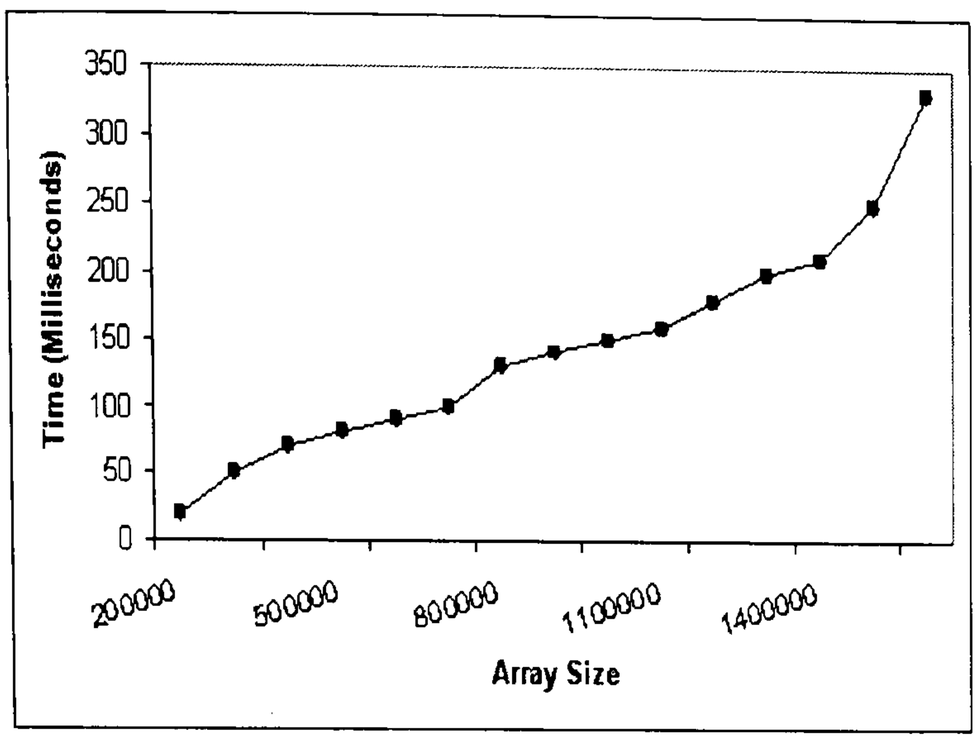



**Figure 8.8: The elapsed time for the Insertion sort Service**

As expected, during the experiment it was noticed that the running time for each service dramatically changes when the size of the array increases. For instance, for the Bubble Sort the suitable dataset interval was from 1500 to 20000, for Quick Sort was from 20000 to 200000 and for the arrays that have more than 200000 elements Insertion Sort Service running time was the shortest.

This feature is exploited to test the self-healing property, that is, the performance drop of a given search algorithm will trigger the recovery, i.e. the self-healing process. To this end, we have implemented a set of QoS rules (policy) to trigger the self-healing process [8]. This means, for instance, if the size of an array exceeds 20000 elements then the *Service Manager* will decide to switch the sort algorithm from the Bubble Sort to the Quick Sort algorithm by invoking the *QuickService,* then the system continues its operation. To measure the system latency due to self-healing, two `starTime` and `endTime` are invoked to measure the start and end clock time respectively of a given self-healing process. This is implemented as follows:

- The discovery process based on Jini discovery with the extensional method to find the service description XML document takes place. On top of the discovery we retrieve the service attributes from the service registry (Fig. 8.9). Each service is registered on the lookup service with attributes to tell the system about the location of the XML document.

```
Public String XmlDocLocation(){
try {
entries = new Entry[2]; // set of attributes

entries =template.attributeSetTemplates;

String nameAttribute =  entries[0].toString());
String xmlDocLocation = entries[2].toString());

return xmlDocLocation;
} catch(Exception e2) {
e2.printStackTrace();
}
```

**Figure 8.9: Retrieving the location of the XML service descriptional document**

- The `XmlDocLocation()` method is called by the java parser class and the XML document is parsed in order to get the name of the method invocation (Fig 8.10).

```
document = builder.parse(xmlDocLocation);
. . .

NodeList mNameNodes =
ServiceDescriptionNode.getElementsByTagName( "MethodName" );

 if ( mNameNodes.getLength() != 0 ) {

 Node MethodName = mNameNodes.item( 0 );

String methodname = (  MethodName.getChildNodes().item( 0
).toString());
                    }
```

**Figure 8.10: An example of getting the invocation method name from an XML document**

- The final step is to pass this method name to a service invocation service that acts as a Jini client [83].

To measure the self-healing process elapsed time, we implemented an example application as described above and ran it 15 times. Each time we passed a different size of array for sorting. The results are shown in Table 8.2:

| Array Size | Time (ms) | Invoked Service |
|---|---|---|
| 500 | 20 | Bubble Sort |
| 2500 | 50 | Bubble Sort |
| 5000 | 70 | Bubble Sort |
| 15000 | 30 | Quick Sort |
| 18000 | 40 | Quick Sort |
| 20000 | 50 | Quick Sort |
| 100000 | 80 | Quick Sort |
| 150000 | 90 | Quick Sort |
| 200000 | 50 | Insertion Sort |
| 250000 | 60 | Insertion Sort |
| 300000 | 80 | Insertion Sort |
| 350000 | 90 | Insertion Sort |
| 1000000 | 180 | Insertion Sort |
| 1500000 | 280 | Insertion Sort |
| 2000000 | 381 | Insertion Sort |

**Table 8.2: The sorting process is performed with the Service manager invoking different services at run-time**

As shown in Table 8.2, using the QoS-like rules when the array size was greater or equal to 15000 elements, the self-healing middleware switched the sorting algorithm from bubble sort to the quick sort service. But when the array size was 200000, the insertion service was activated. Thus, without the self-healing capability the system in case of overload (reaching each service's limits), would either slow rapidly to a halt or would crash instantly.

The following diagram shows the sorting process with and without system self-healing behaviour.

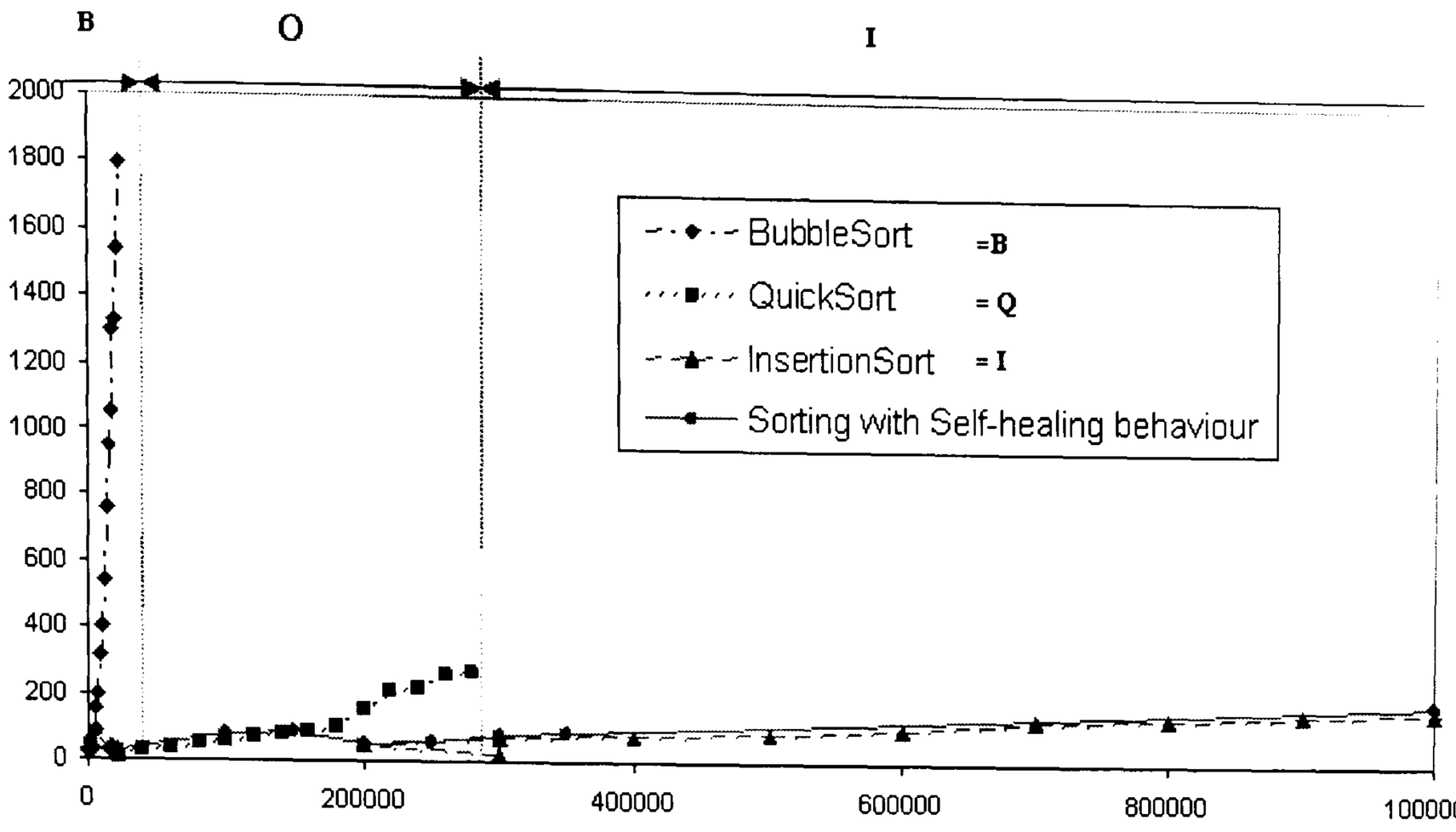



**Figure 8.11: Comparison of the time performance profile in the application with and without Self-healing behavior**

As can be seen from the diagram, regardless of array size, the system continues to perform smoothly without failure. Moreover, the overall time of the system self-healing performance, including the sorting time, is less than 200 ms.

Although this example application does not represent a real case scenario, nevertheless it still demonstrates that if the system is informed of how to behave in critical situations, it can perform self-healing and avoid a crash without human intervention.

### 8.3.3 Home Appliances Scenario

In this experimental application we evaluate the performance of a system with a variable scalability profile. Here we increase the number of activated services, concurrent application services, and different types of applications running in separate virtual containers. As described in Section 6.6, for the home network application prototype, four Jini services were developed, which represent four networked appliances namely: a clock, a lamp, a toaster and a kettle.

For the experiment, the application was run in different situations:

- With self-healing behaviour and without failure of the service in four different cases:
    - Only one service (a clock) was required to be found and invoked.

- Two services were required (a clock and a lamp) to be found and invoked.
- Three services (a clock, a lamp and a toaster) were required to be found and invoked.
- Four services (a clock, a lamp, a toaster and a kettle) were required to be found and invoked.

- To understand how the system would react to service failure and at the same time how efficient and effective it would be in different scales, we deliberately forced a failure of one service (say a clock) and measured the time for recovery for situations when:
  - Only one service (a clock) was running in the system.
  - Two services (a clock and a lamp) were running in the system.
  - Three services (a clock, a lamp an a toaster) were running in the system.
  - Four services (a clock, a lamp, a toaster and a kettle) were running in the system.

As described in previous chapters, the Impromptu application was developed with the ability to allow users to run a number of concurrent applications in different virtual containers. It is predictable that when the load increases the performance will drop. To assess system scalability in this situation, we have run the home appliances application and monitored the system performance in five different situations including;

- The application in Container 1 when there are no other services running in other containers.
- The application in Container 1 when there are five services running in Container 2.
- The application in Container 1 when there are ten services running in Container 2.
- The application in Container 1 when there are fifteen services running in Container 2.

- The application in Container 1 when there are twenty services running in Container 2.

### 8.3.4 The Experimental Results

To evaluate the scalability of the system using the above described experimental scenario, we compared the time overhead of the software system with self-healing and without failure (i.e. just discovery and execution for the first time) against the elapsed time with failure (i.e. time for recovery). Moreover, the elapsed time was measured for different numbers of services (Fig. 8.13). This means we have performed the comparison on the same set of experimental values for the system *with* and *without* performing self-healing. Additionally, the measurement is performed when the applications are running on different virtual containers.

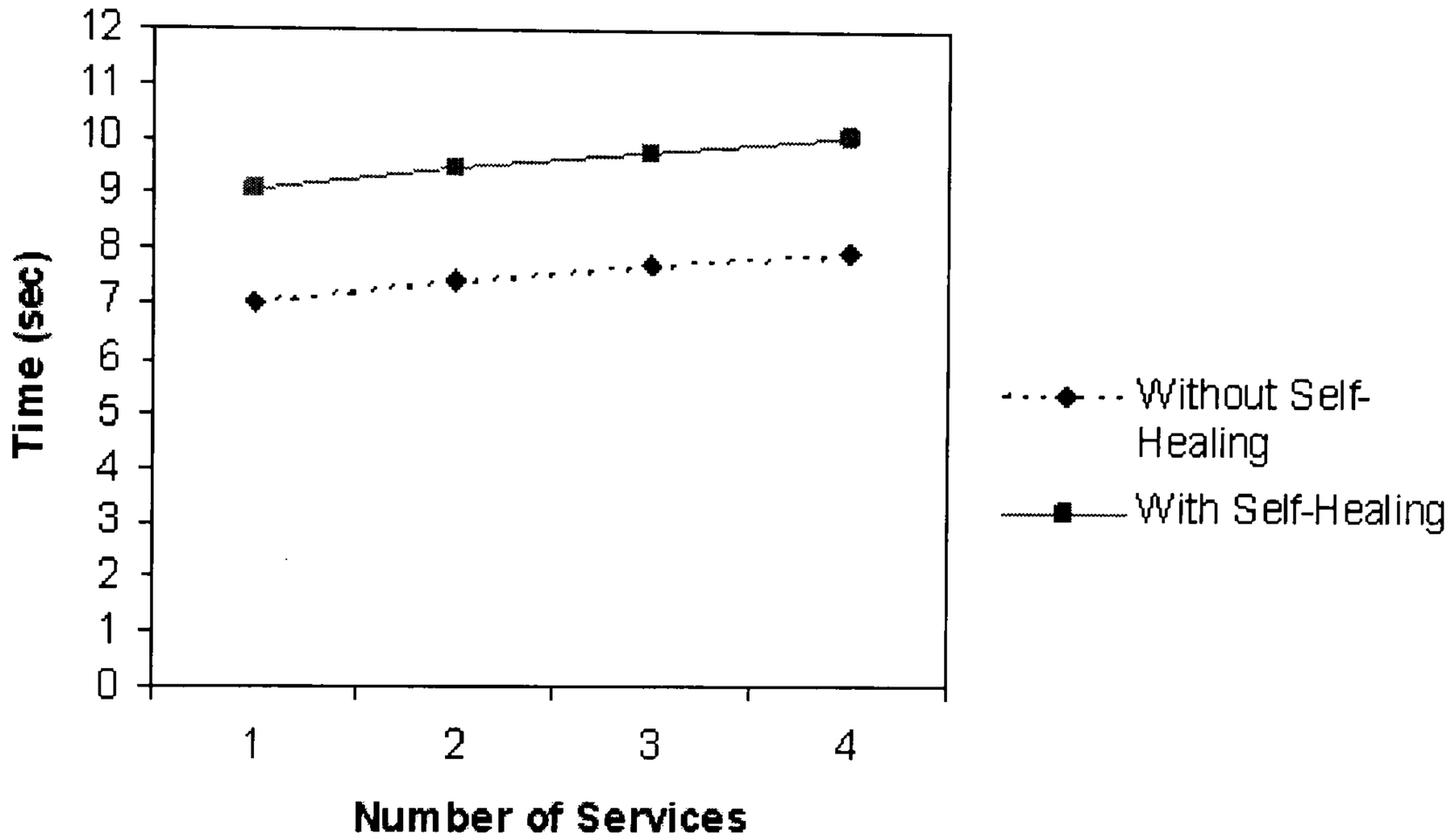



**Figure 8.12: Comparison of the elapsed time for service delivery with and without self-healing**

Figure 8.12 shows the experimental results of comparing elapsed times for the system to perform the discovery and execution of required services and self-healing when the number of those services increases. The x-axis represents the number of required services and the y-axis shows the elapsed time in seconds. This is the time that is taken for the corresponding number of services to be found and executed.

The dotted line represents the time for discovery and execution, initially, without self-healing, while the solid line represents the same time where the self-healing of the system is involved. The graphs are almost parallel because they represent the time

measured for the same number of services with the same attributes. The difference between them is the time that the system takes for failure detection and for the notification to be sent to the services manager to start the recovery process.

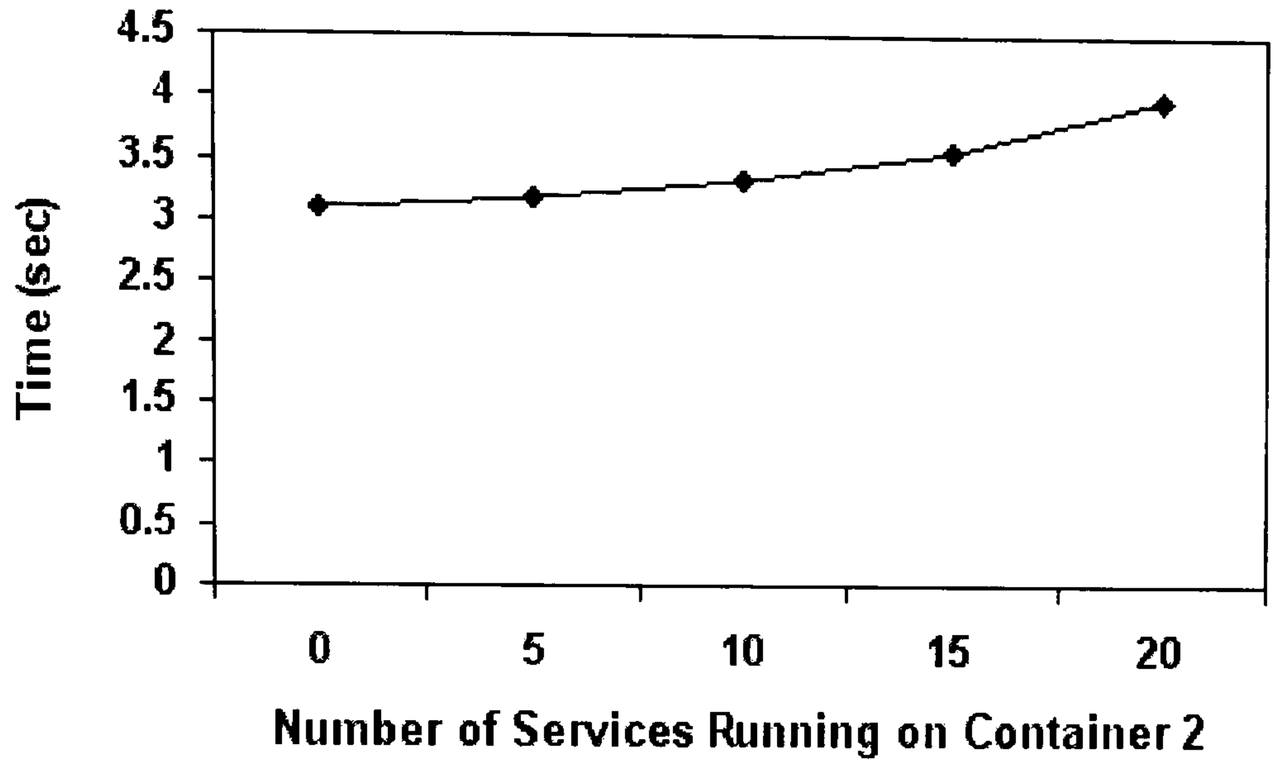



**Figure 8.13: Elapsed time for the application to be executed**

The results of the experiment for this scenario can be summarised as follows:

- The results show that the system consumes more time in conflict (service failure) occurrence situations. However, it is possible to manage the actual recovery time by adding the self-healing feature to the system. It would be difficult to eliminate the recovery time without this feature.
- The experimental results show that the difference between the calculated time of the application with and without self-healing is still reasonable. A large overhead is not added to the running system, as the time increases with increased number of services with and without the additional self-healing feature.
- The comparison between the system not performing and performing the failure detection and self-healing shows that the system continues processing even in the event of conflict (service failure) by managing itself at run time. Without such a feature, the system would need to be shut down or could crash.

Conceptually, to measure the effectiveness of our approach, we examined the average latency (the elapsed time that is required for the system to operate) of the system with the self-healing feature to determine the length of the delay because of this behaviour.

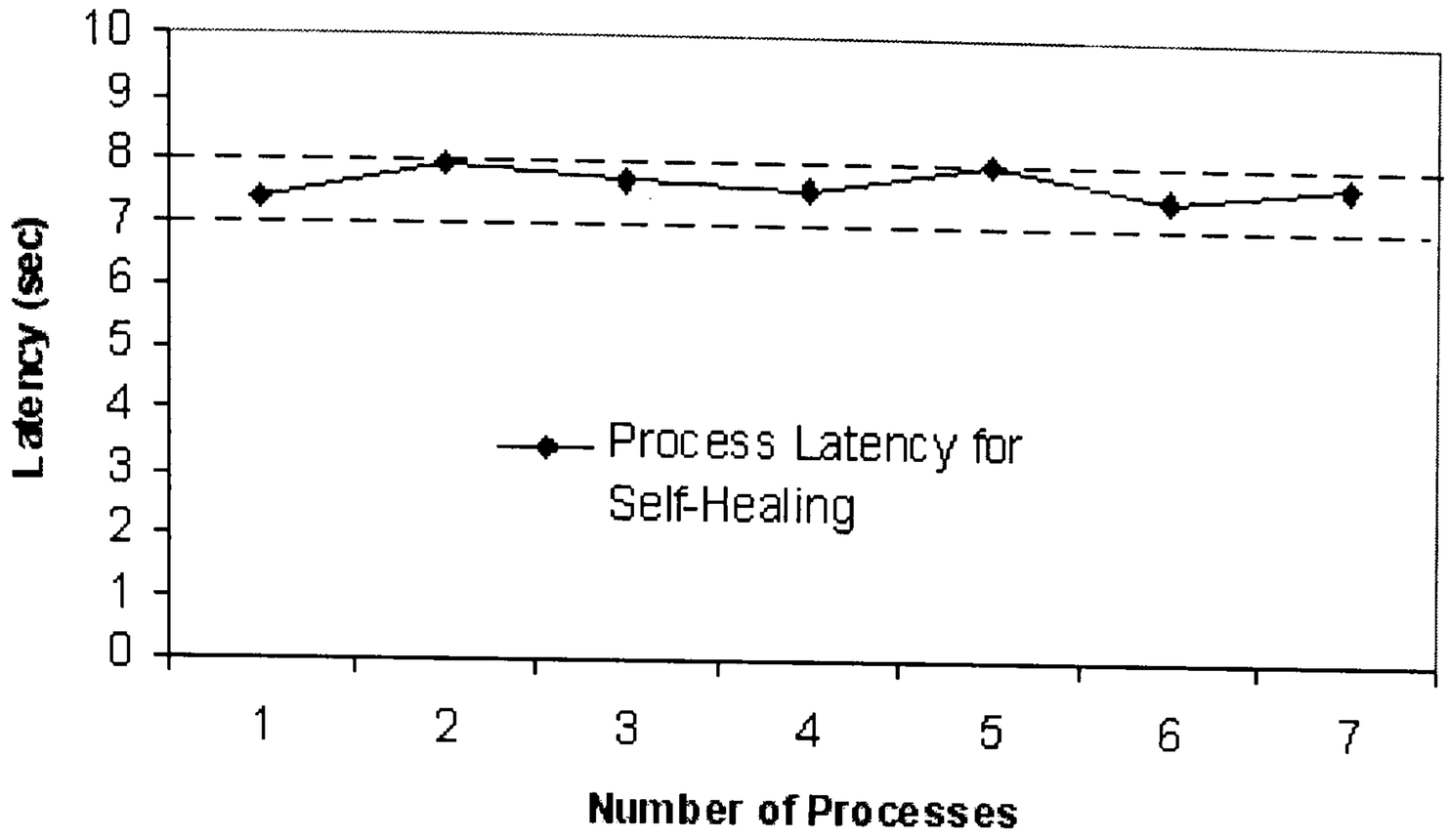



**Figure 8.14: The average latency for self-healing processes in Home Appliances example**

Figure 8.14 shows the average latency for the self-healing process in the Home Appliances example. To calculate the average latency different cases of the self-healing process were run and monitored a number of services, each case was different, chosen randomly. We measured the time without self-healing and with self-healing behaviour of the system for the same number of services in seven different cases. Experiment showed that the time taken for self-healing is between 7 and 8 seconds meaning that latency is not more than 1 second, which is acceptable.

### 8.3.5 Performance of Failure Detection Mechanism

As mentioned in Chapter 4 for failure detection two mechanisms that we put forward for consideration are: Pre-emptive detection and On-use detection. With pre-emptive detection, the system manager checks, on a regular basis, that each of the services associated with the application is alive. If a service fails to respond to the service manager, it is assumed that the service has failed and the recovery process is started and the system manager then notifies the assembly service. With the on-use detection, the system manager monitors locally the service requests and, if a request times-out, it is assumed that the service has failed and the recovery process is started and the system manager then notifies the assembly service.

The performance considerations in this work relates to how these two mechanisms impact on service replacement waiting time and on network traffic. The notion of service replacement waiting time is important: It is the amount of time the application

is prevented from using the service, because the service is found to have failed and is being replaced. The main advantage of the pre-emptive detection is that, as the system manager periodically polls the services, services may be found to be faulty prior to the moment when the application would wish to use them, therefore they can be replaced with zero replacement waiting time.

On the other hand, the pre-emptive mechanism, although reducing the replacement waiting time, generates more network traffic, which may lead to congestion if there are large numbers of applications and services being used by these applications.

Qualitatively, the relative merits of pre-emptive detection and on-use detection are quite clear: With pre-emptive detection, services are monitored regularly, potentially enabling the application to detect a failure prior to the moment when the service would be invoked: Therefore, replacement of that service can be carried out before the service is required for use, so no delay is incurred. However, on closer inspection, the design of such a mechanism is difficult to optimise: If the monitoring frequency is too high, then the system may generate high overheads and network traffic. If the frequency is too low compared to the component failure rate, then it may not be effective, by not detecting faults in time to replace components prior to use. On the other hand, the on-use detection is a simple model that does not attempt to reduce component replacement delay: It assumes that the component is alive and working, and invokes the service when it is required. If the service is down, then the failure is detected, and the recovery process is initiated, and the full service replacement waiting time is incurred.

Even though it is easy to argue the merits of the pre-emptive detection mechanism, quantifying the benefits is not straightforward. In addition to the added complexity of the system, which increases when there are many services in a network and many applications using them, and also the fact that services may be scattered across internetworks, there is the problem of modelling components failure behaviour and service use behaviour.

According to some well-known models available in the literature, component failure frequency follow the bathtub behaviour [117], [118].

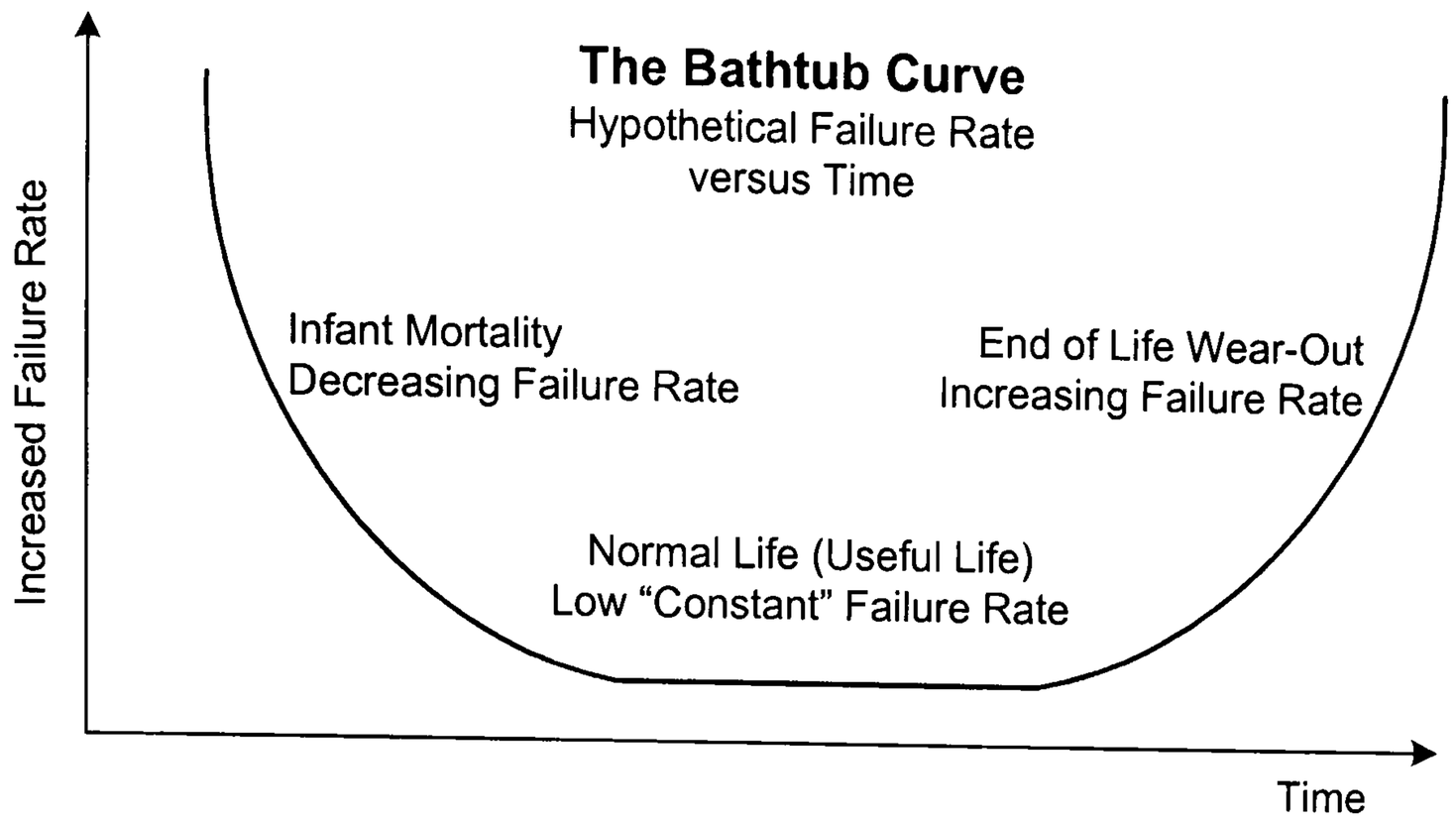



**Figure 8.15: The failure rate of a component over its lifecycle**

This is the failure rate through the life cycle of a component. In normal conditions, we would be considering components with a fixed failure rate, which is often referred to by its inverse, the Mean Time Between Failures (MTBF). It is at this stage that we would consider the components of a service-based application to be operating. Failure can be caused by a variety of events: Machine crash, software error, network disconnection, device hardware failure, etc.

At the constant failure rate stage, inter failure intervals may be modelled according to different distributions. The use of the negative exponential distribution has been proposed in the literature, [117] [118] and we have used it in our model.

### 8.3.5.1 The Experiment

The experiment consists of simulation of a distributed, service-based application, where services fail according to some failure rate (different rates for different services), and following the negative exponential distribution. We make the following assumptions:

When a service fail, if pre-emptive detection is used, the service is replaced by another service providing similar functionality, according to the self-healing behaviour provided in the OSAD model. Overheads are incurred for replacing the service. Once a service that failed has been replaced, the replacement service is subject to failure at the same rate.

If on-use detection is used, we assume that the service remains down after it fails, until an attempt is made to invoke it: As the failure has not been detected by the application until an attempt at using the service takes place, the service is then unavailable.

In order to understand the type of scenario in which each of the above strategies are suitable, we selected, for simulation, two scenarios:

The first scenario is the scenario where an application consists of a large number of services, all of which have a fairly high failure rate. This could, for instance, represent a network of sensors and similar small devices, interconnected through a combination of unreliable wireless links and fixed networks.

The second scenario represents a more stable environment where services are more reliable, having lower failure rate and being connected through a more reliable fixed network infra structure.

In addition to that, for both scenarios, we assume the application invokes the services on a regular basis, for instance to monitor temperature, take a pressure reading or similar action.

### 8.3.5.2 First Scenario

For this simulation, the following setting was adopted:

Number of Services: 15

Distribution of Interfailure Interval: Negative exponential, with mean values $MTBF_{i,}$:

$MTBF_1$: 5 Mins,

$MTBF_{i+1} = MTBF_i + 5$ Mins

The simulation was allowed to run for 15 hours.

Service invocation frequency: 1 invocation of each service, every 5 Mins.

For the pre-emptive detection scheme, the service manager monitors each service also on a regular basis: 2 monitoring periods were chosen: 1 minute and 2.5 minutes.

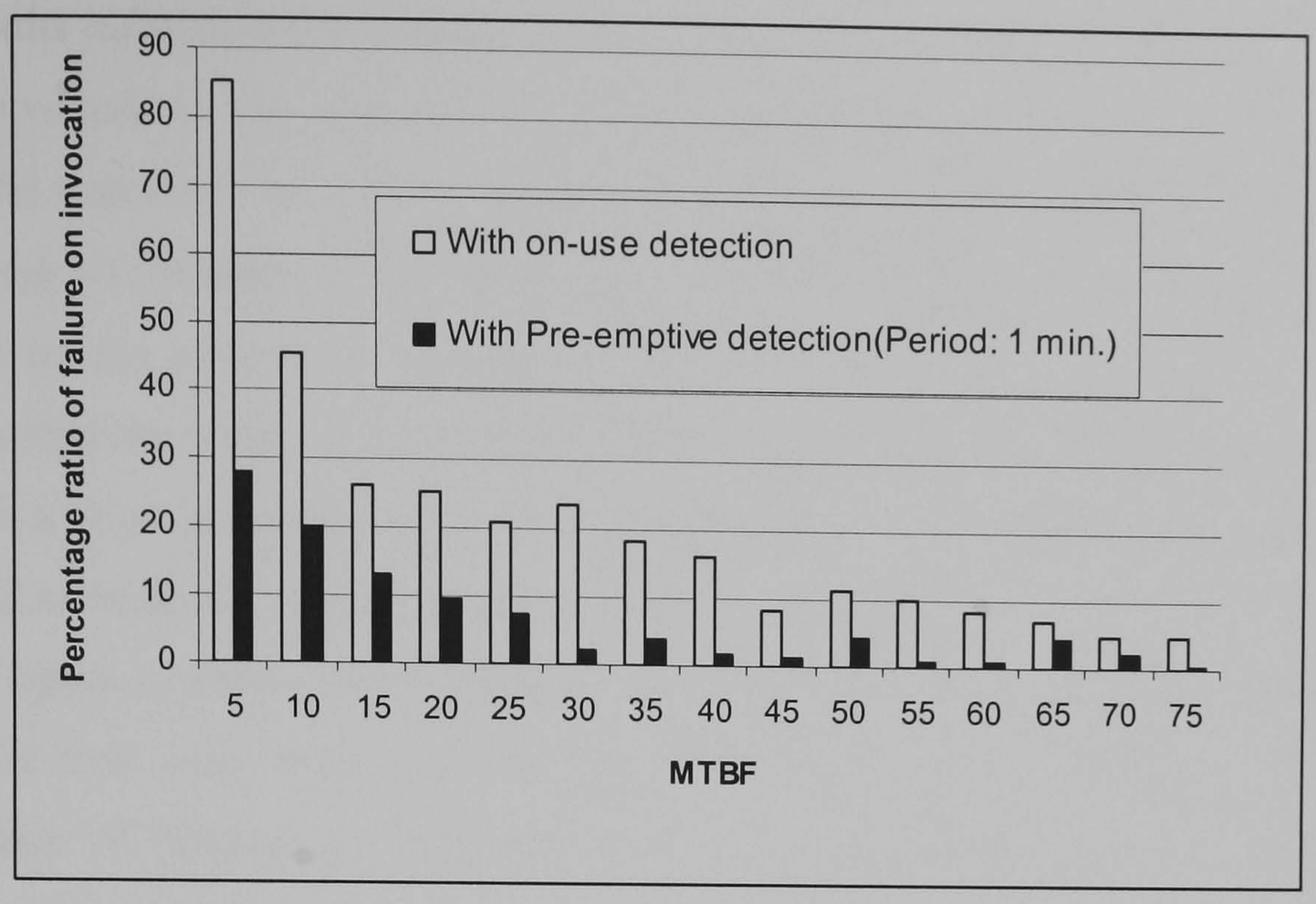



**Figure 8.16: The ratio, in percentages, of failed service invocation to total number of service invocation, as a function of the MTBF, for a system with pre-emptive failure detection (monitoring period = 1 minute) and on-use failure detection.**

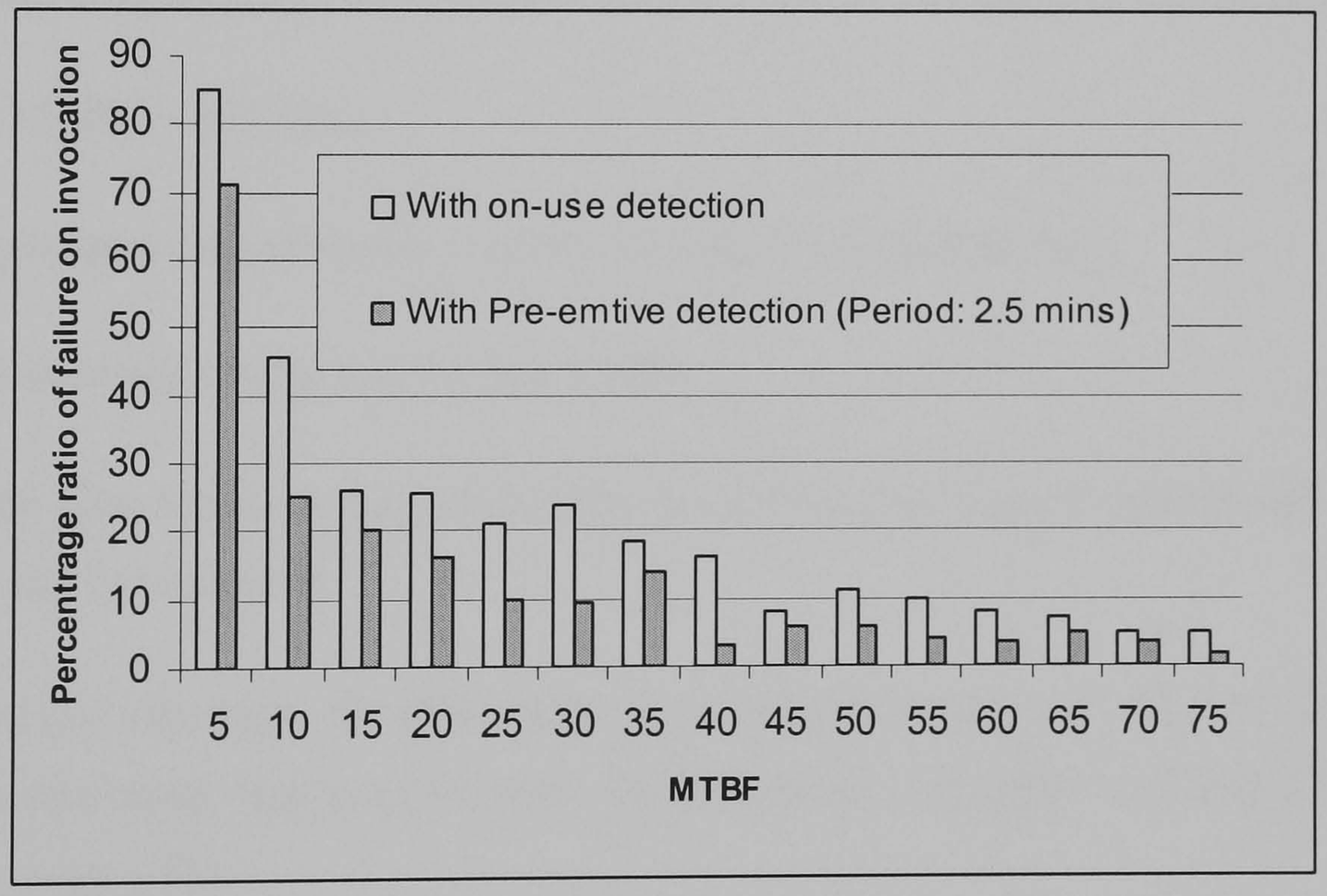



**Figure 8.17: The ratio, in percentages, of failed service invocation to total number of service invocation, as a function of the MTBF, for a system with pre-emptive failure detection (monitoring period = 2.5 minutes) and on-use failure detection.**

The results indicate the percentage ratio of the number of times the service was down when invoked, to the total number of service invocation, with both schemes. As would be expected, for a given invocation rate, the larger the MTBF, the lower the percentage of invocations that fail. However, it is the frequency of failure monitoring, relative to the invocation frequency, which determines the percentage of failed services that are detected successfully. From figures 6 and 7, we see the obvious fact that, for a fixed invocation rate, the higher the failure rate, the higher the percentage of failed invocations. The white bars in both graphs show the same values. The black bar in figure 6 shows the percentage of failures not detected, when pre-emptive detection was used, with a period of 1 min. The grey bar in figure 7 show the percentage of failures not detected, when pre-emptive detection was used, with a period of 2.5 minutes.

### 8.3.5.3 Second Scenario

For this simulation, the following setting was adopted:

Number of Services: 1

Distribution of Interfailure Interval: Negative exponential, with mean values $MTBF_{1,}$:

MTBF1: 300 Mins.

The application was allowed to run for 50 hours of simulated time.

Service invocation frequency: every 5 Mins.

As in the first scenario, the pre-emptive fault detection scheme periods use were: 1 minute and 2.5 minutes.

With on use detection, the percentage of failed invocations were 1.5 %. With pre-emptive detection, with period 1 min., the percentage was 0.68% and with 2.5 minute period it was 1.2%.

This reinforces the conclusions from the first scenario: With very low failure rate, compared to the service invocation rate, it is very unlikely that a service will fail in the first place, so improvement in failure detection is relatively small by using pre-emptive detection

### 8.3.5.4 Evaluation of the experiment

The experiment made a number of assumptions:

- Inter-failure intervals distributions were assumed to be negative exponential. For systems where there are many possible, independent causes of failure, other distributions may provide closer approximations than the exponential.
- Service invocations were assumed to take place at regular intervals. This provides a good model in cases such as monitoring physical values, e.g. temperature and pressure, or any other services that are invoked regularly.

Overall, the experiment provides an approximated understanding of the issue of failure monitoring, and some guidance as to the range of usability of the different schemes proposed.

## 8.4 The Qualitative Evaluation

This section describes a general software evaluation of the On-Demand Service Delivery (OSAD) model. This was conducted to test and evaluate the general functional specifications of the prototyped software and associated applications.

To this end, a number of example scenarios were conducted including: Service-oriented software application example, Home appliances and Sorting Algorithms to assess the model using a set of qualitative metrics such as:

- Functionality, by enabling the distributed application service to be delivered on demand, monitored and with self-healing in the case of failure at runtime.
- Generality, by designing a general structural capability that is not bounded or limited to any specific system or case study and not tied to any specific programming language or technology. And also, by describing the On-Demand Service Assembly and Self-Sealing software pattern language
- Flexibility, by reducing the complexity that could be embedded in any running system for on-demand service delivery, execution and management.
- Extensibility, by referring to the control autonomic middleware, as it can be combined with the OSAD model to fulfil requirements of autonomic, self-controlling, self-healing, self-managing systems.

## 8.5 Meeting the Requirements and Hypothesis of this work

Using the stated hypothesis of the research (Chapter 1, pg. 3) and the general requirements of the Impromptu framework (Chapter 3, pg. 41), this section will outline the review of the framework against the specified requirements and research hypothesis.

| Req. # | Comments |
|---|---|
| **R-1** | In chapter 6 and 8 we have demonstrated that the framework is compliant with existing Service-Oriented Standards. |
| **R-2, R-3** | In chapter 6 we described two prototype applications namely: home appliances and software applications examples, where we have demonstrated that the Framework allows services to be provided, located and used by either a user or software components. We have tested these properties only local area network. However, all described example applications use Jini as a supporting middleware technology and Jini middleware services run over wide area network, therefore using our framework it will be possible for the services to be provided, allocated and used over the wide area network too. |
| **R-4** | In home appliances example we have demonstrated that services are not only discovered, but also invoked. This property is also well demonstrated in sorting algorithms example where we measured time for invocation of each service. |
| **R-5, R-9** | The framework provides an XML-based service and application's assembly service description document, which is automatically generated for general service self-management purpose that is provided by the System Manager middleware service (Chapter 6). In this chapter (Chapter 8) the evaluation indicated a self-healing "latency" value, which is here limited to service discovery time, meta-model (XML document) parsing, invocation time, and application service repair – which is limited |

| | |
|---|---|
| | to hot swapping only, that is replacing failed service by discovered service. |
| **R-6, R-7, R-8** | Using two example applications (home appliances and software services) in Chapter 6, we have demonstrated that the requirements defined in the beginning of this work were met, as: the Framework does provide a mechanism for assembly (Chap. 6); the Framework does support the dependency definition between the services (Chap. 6); and the framework does provide a virtual space for the services to be executed to live (Chapter 6). |
| **R-10, R-11, R-12** | In this chapter we evaluated a System Manager's ability to perform a self-healing process (limited to hot-swapping only), which is triggered via an embedded failure detection (via heart beat monitor) or a request for self-healing service often dispatched from the Monitoring middleware Service [43]. In the current version of the Framework, the self-healing process includes: (i) service discovery (finding and alternative service), service hot-swapping, and update of container metamodel. |

The hypothesis stated in Section 1.3 addressed three main aspects:

- *Self-healing properties can be incorporated in distributed services-based applications through a combination of suitable middleware components*. The Impromptu framework was shown to provide self-healing properties to service based applications: This was reinforced by the case studies discussed in this chapter, where two different distributed service-based applications were shown to recover from failure due to the support provided by Impromptu Framework. The Impromptu service is provided as a combination of core middleware components (Assembly Service and System Manager), in addition to base support from existing middleware (Jini).

- *These components will provide additional functionalities to existing middleware components*. The core services provided by Impromptu are not available in other existing middleware technology, therefore Impromptu

provides additional functionalities (Failure Detection & Recovery, and Service Assemblage)

- *Jini, as a service-based technology, provides a suitable service set to support these components, which will be implemented over Jini as a proof of concept.* The Impromptu prototype was developed using the base services provided by Jini. Jini provided base services such as registration of services, lookup service discovery and service lookup. In this respect, Impromptu can be seen as extending Jini capabilities.

## 8.6 Discussion

It is a significant challenge to produce conclusive evidence of the benefits, merits, effectiveness, correctness and completeness of the proposed model. However, both the qualitative and quantitative evaluations provide positive indications that the proposed model and associated programming model seem to fulfil the defined requirements, and are generic and flexible enough to support the rapid development of a range of distributed applications. In particular, in this evaluation, based on a number of experimental actions, the assembly services succeeded in finding and invoking a distributed service and moreover the service manager performed the self-healing of the application in a reasonable time at run time without human intervention. This clearly indicates a degree of stability and robustness in the software.

Furthermore, in the current version of the Framework, a basic self-healing process validation was not provided to monitor the self-healing process itself in case of failure and/or deadlock. This could be achieved via a dispatch call to the Autonomic Control middleware service [110] to supervise a given self-healing process in line with specified self-healing policies. This is an important mechanism to interrupt and/or adjust a given self-healing process including handing the software control to the human user in the event of abnormal system's behaviour. These issues will be addressed in our future work (Section 9.4).

Regarding the experimental exercise in this chapter, we have learned that the self-healing properties provided by Impromptu services can lead to improved performance of distributed service based applications through the replacement of slow services.

We also showed that Impromptu's ability to provide timely recovery from failure copes well with an increase in the number of services per application and the total number of applications, when the total number of services and applications is not very large. Future work will consider the scalability limitation of Impromptu in more detail.

The decision regarding the specific failure detection scheme to be implemented by the Impromptu system manager was also considered: Experiments were set up to consider on-use detection and pre-emptive detection schemes, showing the relative advantages of each. The probabilistic model assumed negative exponential distribution of inter failure times, which has been used by some researchers. However, future work will include other distributions in order to evaluate the impact of specific distributions on the results. Another improvement to the experimental work, which will be carried out as future work, will include the consideration of variability on the time required to find a suitable replacement to a failed service. This is relevant to take into consideration the large scale of the Internet and how services located in very remote networks may take longer to be replaced.

## 8.6 Summary

This chapter presented an evaluation of the proposed approach for developing a software framework offering self-healing middleware services. In this evaluation, we analysed the effect of self-healing behaviour on the prototype software in terms of processing time.

The measurement results indicated that the performance profile of a system with self-healing behaviour is more efficient than a system without it. Although the elapsed time to self-heal (to detect the failure and perform the recovery) 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 system recovery with proposed self-healing middleware is higher than the elapsed time without it, but it guarantees that the system will perform its task without shutting down or disturbing the system, or incurring costs in the case of outright failure.

# Chapter 9

# Conclusions

## 9.1 Motivations and Approach

In complex distributed systems environments, middleware technology has become of vital importance in order to reduce the effort required to develop applications: typical distributed operations and tasks are provided in middleware modules that can be used by applications developers. Autonomic properties should be incorporated at middleware level, so that applications can incorporate such attributes without excessive effort.

Extending existing works in service-oriented software [30] and autonomic computing [37], the main focus of this work has been to investigate the fundamental requirements for a software framework and associated middleware services to develop on-demand application services. The software framework is intended to enable users to access, over the local or global network, services that are dynamically discoverable, reusable and combinable into flexible self-adaptive applications.

## 9.2 Achievements

This work makes a number of contributions towards a better understanding of software self-healing requirements for autonomic distributed software engineering. Such contributions are summarised below: (1) *A software architectural model*: motivated by and grounded in a range of current research on distributed middleware, service-oriented architectures, service on demand concepts and part of autonomic computing for its support of self-healing and self-awareness. In particular, this work contributes in the development of an architectural model for a middleware-based on-demand self- servicing software framework. The framework described in this thesis unites a number of components including: service description, discovery, invocation, configuration, monitoring and response to events, such as service failure. (2) *A mechanism for assembling distributed services regardless of their location* - is a

representation of how a single functionality provided by one service can be combined with other functionalities provided by other services for the purpose of providing a new united functionality to the user. As a part of the framework, this mechanism contains a number of components that represent the novel elements of this type of system. (3) *An Adaptation model* that is responsible for providing a self-healing behaviour to the framework. The system responds to the failure of a service by searching for an alternative service in a local or global network. The self-healing of the system is performed when the new service is found and invoked to replace the failed one. (4) *A Design Pattern Language* for on-demand service assembly and delivery of software with self-healing behaviour.

We believe that after the number of tests that took place during the prototype implementation period, using different application scenarios, we have contributed to the development of a design pattern language for software that performs service discovery, invocation in an ad-hoc manner and, by using monitoring and quick response performs, self-healing. In addition, a full evaluation of the developed prototype was undertaken using both qualitative and quantitative evaluation, which indicated the overall benefits of such autonomic services though they added to overheads and design complexity.

## 9.3 Thesis Summary

This thesis has offered a new software framework for developing and managing distributed applications in a manner that has not been described before. The detailed description of background theories, methods and achievement of this project are presented as follows:

- Chapter 1 introduced the motivations and technical challenges and outlined the proposed approach and main contributions of the work.
- Chapter 2 introduced the required basic background concepts and principles of Distributed Systems including different categories of middleware. Also were described component-based developments and service-oriented architectures. It also provided a brief overview of Autonomic Computing and its architectural concepts.

- Chapter 3 reviewed the state-of-the-art and related work relevant to self-healing systems development and also reviewed existing service-oriented architectures and frameworks separate from distributed component-based frameworks. Then it outlined what further developments are necessary in these areas and highlighted the requirements for such developments.

- Chapter 4 described the overall design of the proposed framework and described the three main services (Assembly Service, Operational Service and the System Manager) and their sub-services. The self-healing behaviour of the model was also described in this chapter.

- Chapter 5 presented the architecture of the system as a three-layered model, indicating that Impromptu middleware services fit between the supporting technologies and middleware layer and the user application layer. It also described the main architecture of the On-Demand Service Assembly and Delivery (OSAD) model.

- Chapter 6 presented the prototypical implementation of the model based on Java API's and Jini middleware. The implementation of three main services: Assembly Service, Operational Service and the System Manager were described in detail, including the sub-services provided by each of them. At the end of the chapter, two illustrative application examples: Home Appliances scenario and Software Services, were presented.

- Chapter 7 documented six patterns that form the pattern language for distributed applications developers. To describe the patterns, Alexander's original style was used. The Viable System Model was described as a model for the autonomic systems.

- Chapter 8 presented a qualitative and quantitative evaluation of the main functionalities of the framework services. For the quantitative evaluation we used two example scenarios, namely Sorting algorithms and the home appliances scenario. Experimental results showed that the proposed system is efficient and effective.

- Chapter 9 provided the thesis summary, and suggestions for further work.

## 9.4 Proposed Further Works

The thesis detailed a proposed and developed software framework and associated middleware services for autonomic software development. The framework was prototyped and successfully evaluated using a number of cases (scenarios), and highlighted opportunities for further work, including:

- To investigate the integration of the current model of self-healing with a deliberative control service, which can draw on recent work related to policy-based management [111], and norm-governed systems [112]. This should extend the *Impromptu* framework to support predictable and verifiable self-healing.
- To investigate the extension of the self-assembly service to support ubiquitous services discovery and invocation for not only Java RMI-based (Jini) services, but also CORBA, Web services and/or JXTA services.
- To investigate scalability issues of the proposed model using large-scale and widely distributed applications.
- To investigate the feasibility of adopting a decentralised (P2P) provision of the basic components of the *Impromptu* Framework namely the System Manager and the Assembly Service. These can improve the scalability of *Impromptu* Framework and associated middleware services. Hence, its application widely-distributed application models, and more importantly studying issues related to distributed versus centralised self-healing models. This can inform the extended development of a VSM-centric approach to autonomic computing blueprint.
- To investigate how to identify similarity using case-based reasoning techniques, and passing context information between similar components of *Impromptu* framework
- To investigate other options for following a component failure, such as rollback to previous version.

# Appendix A

# Distributed Middleware

## CORBA

One of the best example of object oriented distributed middleware is the *Common Object Request Broker Architecture* (CORBA) [16]. CORBA is an open, distributed, object computing infrastructure specified by the OMG [113], a consortium founded in 1989 to promote the adoption of standards for managing distributed objects. Simply stated, the purpose of CORBA is to provide a simple application programming environment that hides the details of using the operating system services, and provide a common application programming environment across multiple computers and operating systems. CORBA is an open standard, and is not bound to an implementation. With CORBA, the OMG's ultimate goal is to achieve object-oriented standardization and interoperability.

The CORBA specification is defined in the context of the *Object Management Architecture* (OMA), an architectural framework for building interoperable, reusable, portable software components, based on open and standard object-oriented interfaces. The OMA reference model identities and characterizes the components, interfaces, and protocols that compose the OMA. It forms a conceptual roadmap for assembling the resultant technologies while allowing different design solutions.

CORBA objects are specified in the OMG *Interface Definition Language* (IDL). This language is purely declarative, and provides no implementation details. It is used to specify the boundaries of a component and its contractual interface with potential clients. The CORBA specification defines how IDL constructs map to programming languages, such as C, C++, or Java. Figure A.1 shows the five major parts of the OMA reference model:

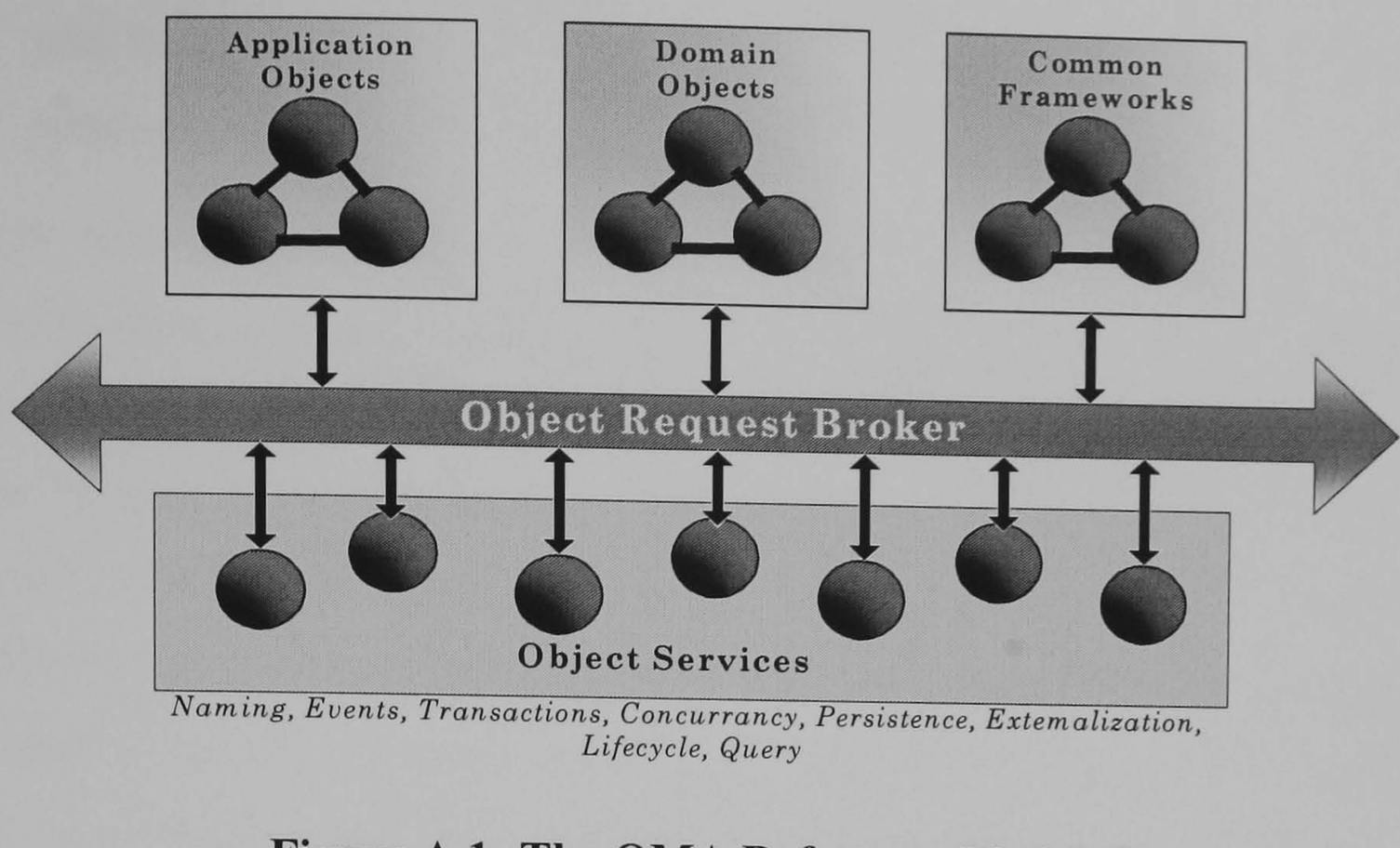



**Figure A.1: The OMA Reference Model [9]**

- The Object Request Broker (ORB), known as CORBA is the communication heart of the standard. It enables object to transparently and reliably invoke operations on remote objects and receive replies in distributed environment. Compliance with the ORB standards guarantees interoperability of objects over a network of heterogeneous systems.

- The Object Services are general-purpose components that are fundamental for developing useful CORBA applications. A CORBA service is a set of CORBA objects with their corresponding IDL interfaces that can be invoked through the ORB. Services are not related to any specific application but are basic building blocks, supporting basic functionalities useful for most applications. Several services have been designed and adopted by the OMG.

- The Common Facilities provide end-user-oriented capabilities useful across many application domains that can be configured to the specific requirements of a particular application. These are facilities that sit close to the user, such as printing, document management, and electronic mail facilities.

- Domain Interfaces represent vertical areas that provide functionality of direct interest to end-users in specific application domains, such as finance or health care.

- Application Interfaces are interfaces specific to end-user applications. They represent component-based applications performing specific tasks for a user. An application is typically composed of a large number of objects, some of

which are specific to the application, and others part of object services, common facilities, or domain interfaces.

The key concept underling the CORBA model is the separation of interfaces and implementations, making it possible to abstract object location and implementation. Objects are not tied to a client or server role: they can act both as clients and as servers. A program is said to be CORBA compliant if it uses only the constructs described in the CORBA specification. An ORB implementation conforms to the CORBA specification if it correctly executes any CORBA compliant program. An application is portable if it can be used with different CORBA implementations (by simply recompiling the source code). Interoperability is the property of several applications running on different CORBA implementations to interact. The CORBA specification defines a standard protocol for ORB interoperability: the Internet Inter-ORB Protocol (IIOP). ORB implementations that use this protocol can interoperate with each other, making it easy to integrate heterogeneous components from different vendors.

The ORB component of CORBA provides more than just messaging mechanisms for remote object invocations. It also provides the environment for managing objects, advertising their presence, and describing their metadata. A CORBA, ORB consists of several pars. These parts are shown in Figure A.2 and described briefly below.

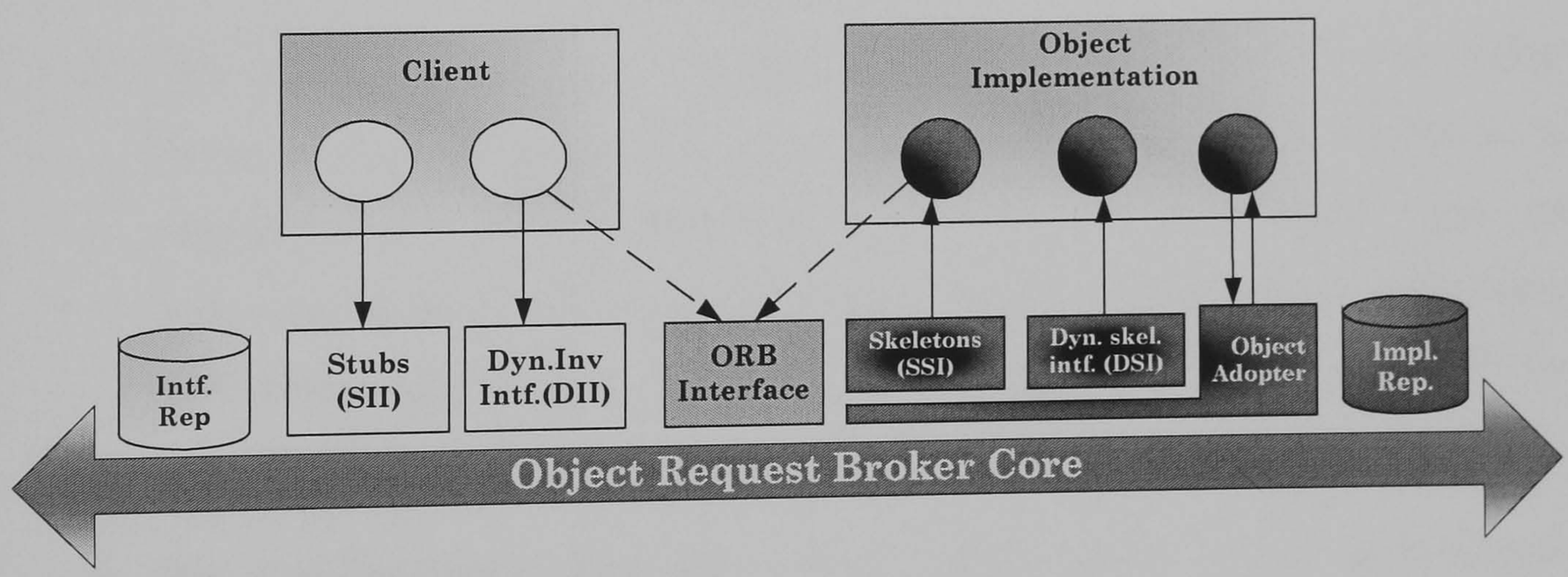



**Figure A.2: The Structure of CORBA, Object Request Broker**

- The Interface Repository (IR) stores interface definitions. It allows the user to obtain and modify the description of component interfaces, the methods that

they support, and the parameters that they require. The interface repository allows CORBA components to have self-describing interfaces.

- The client stubs - or Static Invocation Interface (SII) provide an interface specific API for invoking CORBA objects. A client application can invoke a server object through a client stub. From the client's perspective, the stub is a local proxy for a remote server object. Stubs are generated by an IDL compiler.

- The Dynamic Invocation Interface (DII) allows the creation of requests and invocation of objects at runtime. CORBA defines APIs for looking up the server interfaces, creating requests, generating parameters, issuing the remote call, and getting back the results.

- The ORB interface defines a few general APIs to local ORB services.

- The *server skeletons - or Static Skeleton Interface* (SSI) provide static interfaces to server objects. They contain the code necessary to dispatch a request to the appropriate method. Skeletons are generated by an IDL compiler.

- The Dynamic Skeleton Interface (DSI) provides a runtime binding mechanism for objects that need to handle requests for interfaces not known at compile time. The DSI is the server equivalent of the DII.

- The *Object Adapter* (OA) provides the mechanisms for instantiating server objects, passing requests to them, and assigning them object references. A standard object adapter called the Basic Object Adapter (BOA) must be supported by each ORB implementation. The new CORBA 2.2 specification introduces a new Portable Object Adapter (POA) that overcomes many of the BOA limitations.

- The Implementation Repository stores ORB-specific details about object implementations, their activation policy, and their identity.

## DCOM

Distributed COM is also object-oriented distributed middleware [17]. COM standing for Component Object Model is the technology underlying the various Windows operating systems produced by Microsoft.

DCOM extends the COM to support communication among objects on different computers—on a Local Area Network (LAN), a Wide Area Network (WAN), or even the Internet. Components expose themselves through interfaces and only interfaces. The interfaces are binary which makes it possible to implement the component in a variety of programming languages such as C++, Visual Basic, and Java. A COM component can implement and expose multiple interfaces. A client uses COM to locate the server components and then it queries for a wanted interfaces. The following figure shows how COM establishes the connection between client and server.

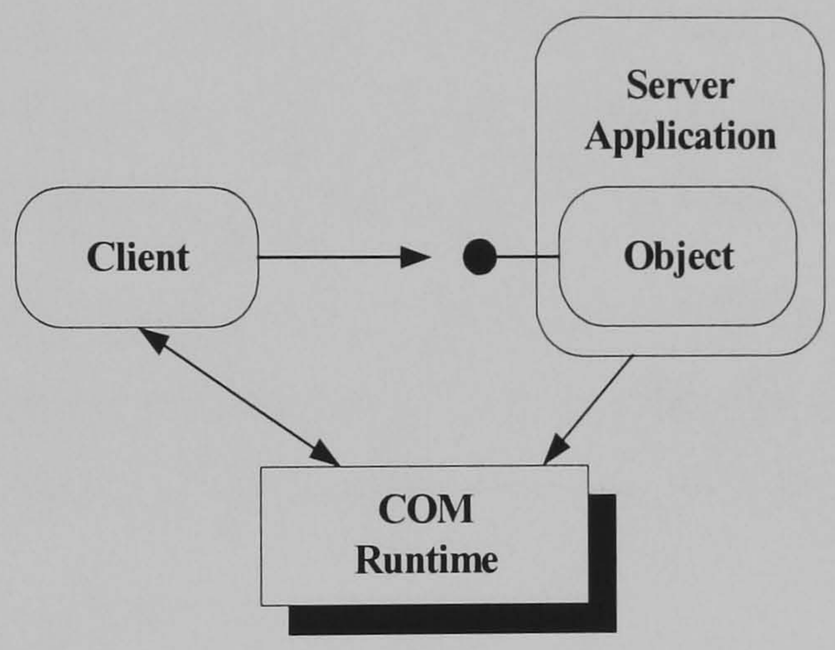



**Figure A.3: COM Interaction**

By defining interfaces as unchangeable units, COM solves the interface versioning problem. Each time a new version of the interface is created a new interface will be added instead of changing the old version. Using DCOM protocol makes COM locations transparent. A client talks to proxy, which looks like a server and manages the real communication with the server.

The overall architecture of DCOM in conjunction with the use of class objects, objects and proxies, is shown in Figure A.3.

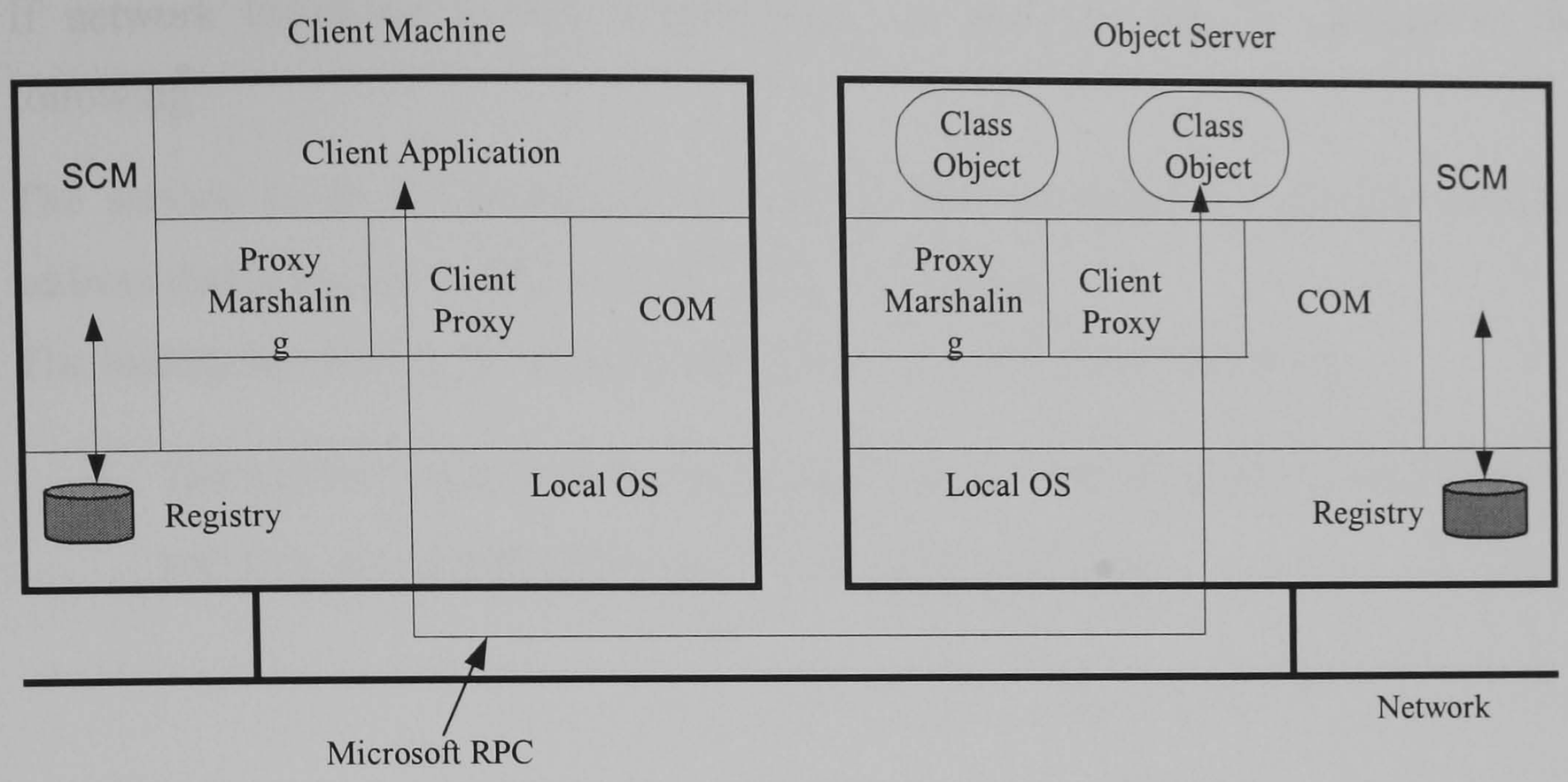



**Figure A.4: The overall architecture of DCOM [9]**

On the client side, a process is given access to the Service Control Manager (SCM) and the registry to help look for and set up a binding to a remote object. The client will be offered a proxy implementing that object's interface.

The object server will consist of a stub for marshalling and unmarshalling invocations, which are passed to the actual object. Communication between the client and server is normally done by means of Remote Procedure Call (RPC).

## The Jini Lookup Service

Each Jini system is built around one or more *Lookup* services. The lookup service is where services advertise their availability so that service clients can find them. There may be one or more lookup service running in the network. When a service is booted on the network, it uses a protocol called *discovery* to find the local lookup services. The discovery protocol will vary depending upon the kind of network. If network broadcast is allowed, then the protocol can be outlined as following [5]:

The service sends a "looking for lookup services" message to the local network. This is repeated for some period of time after initial startup.

1. Each lookup service on the network responds with a proxy for itself.
2. The service registers with each lookup service using its proxy by providing the service's proxy object and any desired initial attributes.

If network broadcast is not possible, then the protocol can be outlined as the following:

The service sends a “looking for lookup service" message to a specific network address that is known to have a lookup service running.
The lookup service on the network host responds with a proxy for itself.

1. The service registers with the lookup service using its proxy by providing the service's proxy object and any desired initial attributes.

# Appendix B

## World Wide Web

The World Wide Web (WWW, W3, the Web) was developed to be a pool of knowledge, which would allow collaborators at remote sites to share their ideas and all aspects of a common project [70]. Physicists and engineers at CERN, the European Particle Physics Laboratory in Geneva, Switzerland, collaborated with many other institutes to build the software and hardware for high-energy physics research. The idea of the Web was prompted by positive experience of a small personal hypertext system used for keeping track of personnel information on a distributed project. The Web was designed so that if it was used independently for two projects, no major or centralized changes would be necessary, but information could smoothly be changed to represent the new state of knowledge. This property of scaling has allowed the Web to expand rapidly from its origins at CERN across the Internet irrespective of boundaries of nations or disciplines.

Today the WWW is essentially a huge client-server system with millions of servers distributed worldwide. Each server maintains a collection of documents; each document is stored as a file. A server accepts requests for fetching the document and transfers it to client. In order to transfer these documents a Uniform Resource Locator (URL) is used. A client interacts with Web servers through a special application called browser. A browser is responsible for properly displaying the document. Most Web documents are expressed by using the special language called HyperText Markup Language, HTML. HTML provides keywords to structure a document into different sections including simple structure elements, such as several levels of headings, bulleted lists, menus and com-pact lists, all of which are useful when

presenting choices and in on-line documents. HTML is an application of the SGML[1] standard [114]. This means that elements used in an HTML document are defined in a Data Type Definition (DTD) following SGML rules. Several extensions expanded the complexity of HTML, so that the World Wide Web Consortium W3C started the XML language initiative.

## XML Technology

With the rapid development of the Web, the limitation of HTML eventually became apparent. The need for a new, standardize, fully extensible, structurally strict language was growing and as a result the Extensible Markup Language (XML) was creating by W3C. XML combines the power and extensibility of its parent language, SGML, with the simplicity demanded by the Web community. The W3C also began to develop XML-based standards for Stylesheet and advanced hyperlinking, and called it Extensible Stylesheet Language (XSL).

HTML documents are human-readable but are not optimised for computer manipulation, whereas most forms of data storage are optimised for computer manipulation and not for human viewing. XML is the first language that makes it both human-readable and computer-manipulable. It is a language of intelligent document that rely on format rather than structure.

The essential characteristics of XML are the data independence, the separation of content and its presentation. Because an XML document describes data, it can conceivably be processed by any application. The absence of formatting instructions makes it easy to parse. This makes XML ideal framework for data exchange. Integration of XML to any applications makes applications more dynamic and interoperating. Because XML's semantic and structural information enables it to be manipulated by any application, much of the information and the operations that were limited only to servers now it can be reachable and performable by client.

Many leading technology developer companies such as IBM, Microsoft or Sun Microsystems use XML technology for developing frameworks that support development of services on web and message-based communication between them.

---

[1] **S**tandard **G**eneralized **M**arkup **L**anguage

XML is one of the supporting technologies for IMPROMPTU framework. The service description model developed using XML in purpose to pass some necessary data from service side to client side is the one of the notions that will be described in this thesis.

## Web Services

The use of Web Services on the World Wide Web is expanding rapidly as the need for application-to-application communication and interoperability grows. Web services provide a standard means of communication among different software applications, running on variety of platforms or frameworks. The most appropriate definition for web services is believed to be as follow:

> "A Web service is a software system identified by a URL, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in manner prescribed by its definition, using XML based messages conveyed by Internet protocols." [115]

A Web service exhibits the following characteristics:

- It is accessible over Web
- It provides an interface that can be called from another program
- It is registering and can be located through a Web service registry
- It communicates using messages over standard Web protocols.
- It supports loosely coupled connection between systems

Obviously Web service itself cannot have all these capabilities and/or functionalities. A variety of technologies that support Web services have been developed. The leading candidates include Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) [79], Universal Description, Discovery, and Integration (UDDI).

## The Language of Web Services

Software or hardware services or components that make these services distributed across the Web share the same rules for service discovery, description, and information exchangeability.

Gartner Research has identified five initial requirements for a Web Service platform:

1. **Discovery**: The mechanism by which services make themselves known and are discovered.
2. **Description:** The way in which specifications are made for what information is passed into and out of the service.
3. **Transport**: The communication method between the user of the service and the service itself.
4. **Environment:** The runtime in which the service executes – the e-business platform describes the environment.
5. **Event Notification:** A mechanism by which a service can be invoked as part of a series of event notification queues.

Different standards have been created to address these requirements.

**UDDI** (Universal Description, Discovery and Integration) is an initiative proposed by Microsoft, IBM and Ariba [79] to develop a standard for an online registry, and to enable the publishing and dynamic discovery of Web services offered by business. UDDI allows programmers and other representatives of a business to locate potential business partners and form the relationships on the bases of the services they provide.

The primary target of UDDI seems to be integration and at least semi-automation of business transactions in B2B e-commerce applications. It provides registry for registering businesses and the services they offer. These are described according to an XML schema defined by UDDI specification. A Web service provider registers its advertisements along with keywords for categorisation. A Web service user retrieves advertisements out of the registry based on keyword search. The UDDI search mechanism relies on pre-defined categorisation through keywords and does not refer to the semantic content of the advertisements. The registry is supposed to function in fashion similar to white pages or yellow pages, where businesses can be looked up by name or by a standard service taxonomy as is already used within the industry.

UDDI aims to facilitate the discovery of potential business partners and the discovery of services and their groundings that are offered by known business partners. This may or may not be done automatically. When the discovery occurs, programmers affiliated with the business partners program their own systems to interact with the services discovered.

Currently, UDDI does not provide or specify content languages for advertisement. Although, the Web Services Description Language (WSDL) is most closely associated with UDDI as the language for describing interfaces to business services registered with a UDDI databases. WSDL is technology also developed by Ariba, IBM and Microsoft. It defines services as collections of network endpoints or ports. In WSDL the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions of messages, which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitute a reusable binding. A port is defined by associating a network address with a reusable binding; a collection of ports define a service. And, thus, a WSDL document uses the following elements in the definition of network services:

- **Types** – a container for data definitions using some type system.
- **Message** – an abstract, typed definition of the data being communicated
- **Operation** – an abstract description of an action supported by the service
- **Port Type** – an abstract set of operations supported by one or more endpoints
- **Binding** – a concrete protocol and data format specification for a particular port type.
- **Port** – a single endpoint defined as a combination of a binding and a network address.
- **Service** – a collection of related endpoints.

To this and WSDL is a template for how services should be described and bound by clients.

**Simple Object Access Protocol** (SOAP) is a lightweight, extensible, XML-based protocol for information exchange in a decentralized, distributed environment. Primarily, SOAP defines a framework for message structure and a message-processing model. SOAP also defines a set of encoding rules for serializing data and a convention for making remote procedure calls. The SOAP extensibility model provides the foundation for a wide range of composable protocols running over a variety of underlying protocols like HTTP. SOAP has been designed to be simple and

extensible. It defines a message-processing model but does not itself define any application semantics, such as programming model or implementation-specific semantics. Although SOAP may be used as a foundation for building complex systems, most features from traditional message systems and distributed object systems are not part of the core SOAP specification.

## Publications by the Author

### 2001

1. M. Allen, **E. Grishikashvili**, N. Badr, A. Taleb-Bendiab, **"Adaptation Engine: an Agent-Based Framework for ad-hoc Service Life-Cycle Management for Meta-Computing"**, *Proceedings of the AISB '02 Symposium on AI and Grid Computing,* Imperial College of Science, 2001.

### 2002

**2._ E. Grishikashvili,** M. Allen, A. Taleb-Bendiab, **"IMPROMPTU: On-Demand Self-Servicing Software Framework",***Proceedings of 2nd International Workshop on intelligent Systems Design and Applications, ISDA 2002, Atlanta - USA, August 2002.*

**3. E. Grishikashvili,** N. Badr, M. Yu, D. Reilly, A. Taleb-Bendiab, **"Autonomic Computing: A Service-Oriented Framework to Support the Development and Management of Distributed Applications",** *Proceedings of 3rd Annual Postgraduate Symposium on the Convergence, Telecommunication, Networking and Broadcasting. PgNet, Liverpool, June 2002.*

### 2003

**4. E. Grishikashvili,** N. Badr, A. Taleb-Bendiab, **"Service-Oriented Approach for Distributed application Assembly and Management",** *Proceedings of 4th Annual Postgraduate Symposium on the Convergence, Telecommunication, Networking and Broadcasting. PgNet, Liverpool, June 2003.*

**5. E. Grishikashvili,** N. Badr, D. Reilly, A. Taleb-Bendiab, **"From Component-Based to Service-Based Distributed Applications Development and Life-Time Management ",** *Proceedings of 29th Euromicro Conference – "Component-Based Software development", Antalya - Turkey, September 2003.*

**6.** M. Yu, **E. Grishikashvili**, D. Reilly, A. Taleb-Bendiab **"Polyarchical Middleware for On-Demand and Multi-Standard Services' Composition for Ubiquitous Computing",** *UNITN2003 The New Computing Paradigm for the Networked World. Trento - Italy, December 15-18, 2003.*

### 2005

**7. E. Grishikashvili-Pereira**, R. Pereira, A. Taleb-Bendiab, **"Performance Evaluation for Self-Healing Distributed Services",** *Proceedings of the First International Workshop on Performance Modelling in Wired, Wireless, Mobile Networking and Computing (PMWMNC-2005), with IEEE International Conference on Parallel and Distributed systems, Fukuoka, Japan, July 20-22, 2005.*

**8. E. Grishikashvili-Pereira,** R. Pereira, A. Taleb-Bendiab, **"Fault Detection Mechanism for Autonomic Distributed Applications."** *Proceedings of 21st Annual UK Performance Engineering Workshop. UKPEW 2005, Newcastle, UK. 14-15 July, 2005*

**9. E. Grishikashvili Pereira,** R. Pereira and A. Taleb-Bendiab, **"Performance evaluation for self-healing distributed services and fault detection mechanisms".** Journal of Computer and Systems Science, ELSEVIER, March 2006.

## References


1. IEEE, *6th IEEE International Conferance on Engineering of Complex Computer Systems (ICCECS 2000)*, in *IEEE Computer Society*. 2000: Tokyo, Japan
2. A. Laws, A.T.-B., and S. J. Wade, *Towards a Viable Reference Architecture for Multi-Agent Supported Holonic Manufacturing Systems.* International Jurnal of Applied Systems Science, 2001. Vol. 1.
3. G. Bieber, J.C., *Introduction to Service-Oriented Programming*, in *Motorola ISD*. 2002
4. I. Crnkovic, M.L. *A Case Study: Demands on Component-Based Development*. in *22nd International Conference on Software Engineering*. 2000: ACM Press.
5. *Jini Technology*. http://www.jini.org
6. Newmarch, J., *Remote Events*, in *A Programmer's Guide to Jini Technology, chapter 14: Remote Events*. 2000, Apress: USA.
7. *Java remote MethodInvocation - Distributed Computing for Java*. http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html
8. Badr, N. *An Investigation into Autonomic Middleware Control Services to Support Distributed Self-Adaptive Software* Schoool of Computing and Mathematical Scinces Liverpool John Moores University Liverpool 2003
9. A. S. Tanenbaum, M.V.S., *Distributed Systems Principles and Paradigms*. 2002: Prentice Hall. 0-13-088893
10. G. Coulouris, J.D., T. Kindberg, *Distributed System, Concepts and Design*. Third Edition ed. 2001: Addition Wesley. 0201-61918-0
11. G. Coulouris, J.D., T. Kindberg, *Distributed System, Concepts and Design*. Second Edition ed. 1996: Addition Wesley. 0201-61918-0
12. *Principles of Distributed Middleware*. http://www.cs.huji.ac.il/~davb/huji/pdm/links.html
13. Celko, J., *SQL for Smarties: Advanced SQL Programming*. Expanded 2nd Edition ed. 1999: Morgan Kaufmann. 1558605762
14. Scientific Computing Associates *LindaSpace*. 2003. http://lindaspaces.com/products/linda.html
15. Addition-Wesley Inc, Argonne National Laboratory *Designing and Building Parallel Programs, Tuple Space*. http://www-unix.mcs.anl.gov/dbpp/text/node44.html
16. OMG *CORBA Basics*. http://www.omg.org/gettingstarted/corbafaq.htm
17. Carnegie Mellon University *Component Object Model (COM), DCOM and Related Capabilities*. 1997. http://www.sei.cmu.edu/str/descriptions/com_body.html
18. P. Sliviter *Object Linking and Embedding*. 1998. http://www.it.bton.ac.uk/Research/Archive/noolearning/facilits/exemplars/oovb/schedule/section17/ab24_3.htm

19. Carnegie Mellon, Software Engineering Institute *Copmonent Object Model*. 1997. http://www.sei.cmu.edu/str/descriptions/com_body.html

20. B. Sheresh, D.S., R. Cowart, *Microsoft Transaction Server*, in *Microsoft Windows Nt Server Administrator's Bible: Option Pack Edition*. 1999, IDG Books.

21. Loughry, M., *Active Directory for Dummies*. 1999: For Dummies; Bk&CD-Rom edition. 0764506595

22. Sun Microsystems *Jini Network Technology*. http://wwws.sun.com/software/jini

23. Erl, T., *Service-oriented Architecture: A Field Guide to Integrating XML and Web Services*. 2004: Prentice Hall PTR. 0131428985

24. *Java 2 Platform, Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/

25. Microsoft Corporation *Microsoft .NET*. http://www.microsoft.com/net/

26. P. Coad, J.N., *Object-Oriented Programming*. 1st Edition ed. 1993: Pearson Education. 013032616X

27. Bieber, G. *Openwings - Closing the Personal Digital Divide*, Motorola 2001.

28. J. Carpenter, G.B. *Openwings - Component Service Specification*, General Dynamics Decision Systems 2002. Alpha Ver 0.72.

29. E.Grishikashvili, N.B., D. Reilly, A. Taleb-Bendiab. *From Componen-Based to Service-Based Distributed Applications Development and Lif-Time Management*. in *EuroMicro*. 2003. Antalya, Turkey.

30. Pallos, M.S., *Service-Oriented Architecture: A Primer*. EAI Journal, 2001.

31. I. Crnkovic, M.L. *Component-Based Engineering - New Paradigm of Software Development*. in *MIRPO 2001*. 2001. Opatija, Croatia.

32. Han, J. *An Approach to Software Component Specification*, Monash University 1999. Melbourne, Australia,

33. Sziperski, C., *Component Software - Beyond Object-Oriented Programming*. Second Edition ed. 2002: Addison-Wesly and ACM Press. 0-201-74572-0

34. A. W. Brown, K.C.W. *An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering*. in *ICSE 98*. 1998. Japan.

35. School of Computer Science, Carnegie Mellon University *Architecture Description Language*. 1998. http://www-2.cs.cmu.edu/~acme/acme_documentation.html

36. Emmerich, W., *Engineering Distributed Objects*. 2000: John Wiley & Sons Ltd. 0-471-98657-7

37. *Autonomic Computing*. 2000. http://www.research.ibm.com/autonomic/

38. J. O. Kephart, D.M.C. *The Vision of Autonomic Computing*, IBM Tomas J. Watson Research Center 2003.

39. Corporation, I. *Introduction to Autonomic Computing*, IBM Corporation, Software Group 2001. Somers, NY,

40. Koopman, P. *Elements of the Self-Healing System Problem Space*. in *ICSE WADS03*. 2003. Portland.

41. M. Mikic-Rakic, N.M., N. Medvidovic. *Architectural Style Requirements for Self-Healing Systems*. in *Wass'02*. 2002. Charleston, South Carolina, USA.

42. P. N. Gross, e.a. *An Active Events Model for Systems Monitoring*. in *Complex and Dynamic System Architecture*. 2001. Brisbane, Australia.

43. D. Reilly, A.T.-B., A. Laws, N. Badr. *An Instrumentation and Control-Based Approach for Distributed Application Management and Adaptation*. in *Woss'02*. 2002. Charleston, South Carolina, USA.

44. D. Garlan, B.S. *Model-Based Adaptation for Self-Healing Systems*. in *WOSS'02*. 2002. Charleston, South Carolina, USA.

45. P. Oriezy, M.M.G., R. N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, *An Architecture-Based Approach to Self-Adaptive Software*. IEEE Intellingent Systems, 1999. 14.

46. *The Rio Project*. 2001. http://developer.jini.org/exchange/projects/rio

47. Microsystems, S. *RIO Architecture Overview*, Java Center 2000.

48. Dearing, R. *Service-Oriented Architecture Using Jini*, SearchWebServices.com 2003.

49. Microsystems. *Open Net Environment (ONE) Software Architecture - An Open Architecture for Interoperable, Smart Web Services*, Sun Microsystems Inc 2001. Palo Alto, CA,

50. Reichardt, D. *A Field Guide to Services On Demand and Sun ™ ONE*, Sun Microsystems 2001. September 2001.

51. Combs, N. *BBN-DASADA, Agent-Based Service-Orented Software Composition*, BBN Technologies 2001. Cambridge,

52. Technologies, B. *Cougaar Architecture Document*, A BBN technologies 2001. Version 8.6. http://www.cougaar.org

53. Sun Microsystems *Openwings*. 2003. http://www.openwings.org/

54. Invent, H.-. *Java Service Framework: An Approach to Open-Standards Services Sevelopment*, 2001.

55. Mochizuki, M. *A Middleware Architecture for Improvised Assembly of Component-Based Application* Graduate School of Media and Governance Keio University Fujisawa, Kanagawa, Japan 2000

56. *Technical Committee on Complexity in Engineering*. http://www.elet.polimi.it/tccx

57. C. Herring, S.K., *Component-Based Software Systems for Smart Environments*. Personal Communications Interactive, IEEE, 2000.

58. M. J. Williams, A.T.-B. *A Toolset for Architecture Independent, Reconfigurable Multi-Agent Systems*. in *Proceedings of First International Workshop on Mobile Agents*. 1998.

59. W. E. Hefley, D.M. *Intelligent User Interfaces*. in *International Conference on Intelligent User Interfaces*. 1993. Orlando, Florida, US: ACM Press.

60. Beer, S., *Diagnosing the System for Organizations*. 1985, Chichester: John Wiley & Sons.

61. Herring, C.E. *Viable Software - The Intelligent Control Paradigm for Adaptable and Adaptive Architecture* Department of Information Technology and Electrical Engineering University of Queensland Brisbane 2002

62. A. Laws, A.T.-B., and S. J. Wade. *Managing Complexity in Self-Adaptive Software: A Cybernetic Approach*, School of Computing and Mathematical Sciences, Liverpool John Moores University 2001. Liverpool,

63. A. Laws, A.T.-B., and S. J. Wade. *From Wetware to Software: A Cybernetic Perspective of Self-Adaptive Software*. in *Self-Adaptive Software, Second Internationals Workshop, IWSAS*. 2001. Balatond, Hungary: Springer.

64. M. R. Geneserth, S.P.K., *Software Agents*. 1994, Communications of the ACM. p. 48-54.

65. L. Bellissard, M.R. *From Distributed Objects to Distributed Components: the Olan Approach*. in *ECOOP Workshop published in Special Issues in Object-Oriented Programming*. 1997.

66. DARPA, BBN Technologies *Quality Object (QuO)*. 2002. http://quo.bbn.com/

67. Sun Microsystems *The Reflection API*. 2004. http://java.sun.com/docs/books/tutorial/reflect/

68. A. Brown, S.J., K. Kelly. *Using Service-oriented Architecture and Component-Based Development to build Web Service Applications*, Rational - the software development company 2002. October. TP032.

69. IBM, RedBooks *Patterns: Flexible Self-Service Applications Using Process Choreography*. 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246322.pdf

70. Lee, B., *Weaving the Web*. 1999: Harper San Francisco. 0062515861

71. Udell, J., *Componentware*. 1994, CMP Media LLC: Byte.com

72. J. Schneider, O.N. *Components, Scripts and Glue*. in *Software Architectures - Advances and Applications*. 1999: Springer.

73. McIlory, M.D., *Mass Product Software Components*, in *Software Engineering*. 1969, P. Naur and B. Randell: NATO Scinece Commitee

74. M. Lumpe, J.S., O. Nierstrasz, and F. Achermann. *Towards a Formal Composition Language*. in *ESEC'97 Workshop on Foundations of Component-Based Systems*. 1997. Zurich.

75. M. Shaw, D.G. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall 1996. April.

76. Pree, W., *Design Patterns for Object-Oriented Software Development*. 1995: Addison-Wesley.

77. E. Gamma, R., Helm, R. Johnson, J. Vlossodes, *Design Patterns - Elelments of Reusable Object-Oriented Software*. 1st Edition ed. 1995: Addison-Wesley Professional. 0201633612

78. Miner, N., *Peer-to-Peer - Harnessing the Power of Disruptive Technologies*. March 2001. 0-596-00110-X

79. Ariba, International Business Machines Corporation, Microsoft *Web Services Description Language (WSDL)*. 2001.

80. Jini Community *Rio Architecture Overview*. http://www.jini.org/projects/rio

81. Cheng, S. *Using Architectural Style as a Basis for Self-Repair*. in *IEEE/IFIP Conference onSoftware Architectures (WICSA2002)*. 2002. Montreal, Canada.

82. Sun Microsystems *Java Technologies*. http://java.sun.com/

83. J. Newmarch *Guide to Jini technology*. 2004. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml

84. Brown, A.W., *Larcge Scale Component Based Development*. 1st Edition ed. 2000: Prentice Hall. 013088720X

85. Tokuda, M.M.a.H., *Improvised Assembly Mechanism for Component-based Mobile Applications*. Mobile Multimedia Communications, 2001. E84-B, No.0.

86. P. Robertson, R.L., and H. Shrobe. *Introduction: the First International Workshop on Self-Adaptive Software*. in *The First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000. Oxford, UK: Springer 2001.

87. E.Grishikashvili, N.B., D. Reilly, A. Taleb-Bendiab. *Autonomic Computing: A Service-Oriented Framework to Support the Development and Management of Distributed Applications*. in *3rd Annual postgraduate Symposium, The Convergence of Telecommunications, networking and Broadcasting, PGNet 2002*. 2002. Liverpool, UK: School of CMS, Liverpool John Moores University.

88. N. Badr, D.R., A. Taleb-Bendiab. *A Conflict Resolution Control Architecture for Self-Adaptation*. in *International Workshop on Architecting Dependable Systems WADS 2002*. 2002. Orlando, Florida, US.

89. Carnegie Mellon *Networked Appliances, Project 19-601*. http://www.andrew.cmu.edu/~msin/19601

90. Smart Home Forum, The HAVi organisation *HAVi*. 2000. http://www.smarthomeforum.com/start/havi.asp?ID=12

91. S. Moyer, D.M. *The Internet Alarm Clock - A Network Appliance Case Study*, Telcordia Technolohies 2000. Morristown, NJ, USA,

92. A. Keller, U.B., G. Kar. *Classification and Computation of Dependencies for Distributed Management*. in *Fith IEEE Symposium on Computers and Communications (ISCC 2000)*. 2000. Antibes - Juan-les-pins, France.

93. Hasselmeyer, P. *Managing Dynamic Service Dependencies*. in *Twelfth International Workshop on Distributed Systems: Operations and Management (DSCOM 2001)*. October 2001. Nancy, France.

94. C. Alexander, S.I., M. Silverstein, *A Pattern Language: Towns, Buildings, Constructons*. Senter for Environmental Structure. 1977: Oxford University Press. 0-19-501919-9

95. Beer, S., *The heart of the Interprise.* 1979, Chichester: John Weiley & Sons.

96. Beer, S., *Brain of the Firm*. 1981, Chichester: John Wiley & Sons.

97. D. S. Schmidt, M.S., H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Networked and Concurrent Objects*. Vol. 2. 2000, New York: Wiley & Sons.

98. M. Fowler, D.R., M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture.* 2002, Reading, Massachusett: Addison-Wesley.

99. Cunningham, W., *The CHECKS Pattern Language of Information Integrity*, in *Pattern Languages of Programm Design*, J.O.C.a.D.C. Schmidt, Editor. 1995, Addison-Wesley: Reading, Massachusetts.

100. Borchers, J., *A Pattern Approach to Interaction Design*. 2001, New York: Wiley & Sons.

101. IBM, RedBooks *Patterns: Service-Oriented Architecture and Web Services*. 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf

102. Cleeland, D.C.S.a.C., *Appyling Patterns to develop Extensible ORB Middleware.* IEEE Communication Magazine, 1999. 37.

103. Lee, D., *Concurrent Programming in Java: Design Principles and Patterns*. Second ed. 2000, Boston: Addison-Wesley.

104. M. Yu, A.T.-B., D. Reilly, W. Omar. *Multi-Standard Service Interoperation Protocol Through Polyarchical Middleware.* in *4rth Annual postgraduate Symposium, The Convergence of Telecommunications, networking and Broadcasting, PGNet 2003*. 2003. Liverpool, UK: School of CMS, Liverpool John Moores University.

105. M. Yu, A.E.g., Taleb-Bendiab, D. Reilly, W. Omar. *Plolyarchical middleware for On-Demand and Multi-Standard Services' Composition for Ubiquitous Computing*. in *The First International Conference on Service Oriented Computing*. 2003. Trent, Italy: The University of Trent.

106. E.Grishikashvili Pereira, R.p., A. Taleb-Bendiab. *Performance Evaluation for Self-Healing Distributed Services*. in *The First International Workshop on Performance Modelling in Wired, Wireless, Mobile Networking and Computing (PMWMNC-2005), with IEEE International Conference on Parallel and Distributed Systems*. 2005. Fukuoka, Japan: IEEE.

107. *Discussion of Sorting Algorithms*. 1996. http://atschool.eduweb.co.uk/mbaker/sorts.html

108. Sun Microsytems *The Java Tutorial - RMI*. http://java.sun.com/docs/books/tutorial/rmi/

109. R. Helman, D.a.B., J. JaJa. *Parallel Algorithms for Personalised Communication and Sorting with an Experimental Study*. in *ACM Symposium on Parallel Algorithms and Architectures*. 1996. Padua, Italy.

110. E. Grishikashvili, N.B., A. Taleb-Bendiab. *Service-Oriented Approach for Distributed application Assembly and Management*. in *The 4th Annual Postgraduate Symposium on the Convergence, Telecommunication, Networking and Broadcasting. PgNet*. 2003. Liverpool, UK: LJMU.

111. N. Badr, D.R., A. Taleb-Bendiab. *Policy-Based Autonomic Control Service*. in *The Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'04)*. June 2004. Yorktown Heights, New York.

112. N. Badr, M.R., D. Reilly, A. Taleb-Bendiab. *A Deliberative Model for Self-Adaptation Middleware Using Architectural Dependency*. in *The 15th International Workshop on Database and Expert Systems Applications (DEXA'04)*. August 2004. Zaragoza, Spain.

113. *Object Management Group*. http://www.omg.org

114. Cover Pages *Standard Generalized Markup Language*. 2002. http://xml.coverpages.org/sgml.html

115. W3C *Web Services Activity*. 2002. http://www.w3.org/2002/ws/

116. Fremantle, P., Weerawarana, S., Khalaf, R., 2002. Enterprise services. Commun. ACM 45 (10), 77–80.

117. Wilkins, D. J. (2002) The Bathtub Curve and Prodct Failure Behavior. The eMagazine for Reliability Professionals Volume, http://www.weibull.com/hotwire/issue21/hottopics21.htm

118. SemiconFareast. (2004). "Life Distribution." from http://www.semiconfareast.com/lifedist.htm.