DYNAMIC SERVICE RECOVERY IN A GRID ENVIRONMENT

Sihle Sicelo Sibiya

20034232

A dissertation submitted in fulfilment of the requirements for the degree of

Master of Science (Computer Science)

Department of Computer Science, Faculty of Science and Agriculture, University of Zululand

Supervisor: Dr S.S. Xulu

Co-Supervisor: Prof M.O. Adigun

2008

DECLARATION

I, Sihle Sicelo Sibiya, declare that this dissertation represents my work and that it has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from published or unpublished work of others has been acknowledged in the text and a list of references is given.

Signature of student

DEDICATION

I dedicate this work to my loving family, for believing in and encouraging me through the difficult times. None of this would be possible without their love, support and encouragement.

ACKNOWLEDGMENTS

I thank the Lord for all His blessings bestowed on me in seeing this research work through; without which this research could not have become a reality.

I would like to extend my sincere thanks to my supervisor Dr S.S. Xulu for his advice, guidance and fatherly talks. He gave me courage to face the challenges in research.

I extend my special thanks to my co-supervisor, Prof M.O. Adigun, who willingly assisted and guided me right through my research work and for making us become better researchers. I will like to thank him for believing in me and giving me the courage to face up to my uncertainties.

I would also like to thank J.S. Iyilade, O.O. Olugbara, E. Jembere and all the research assistants in the Department for their contribution to this work.

I am indebted to all who have helped me in making this research work a great success in all aspects.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Grid computing is fast becoming a popular technology in both academic and business environment. The adoption of this technology into the business environment has been slow due to some challenges such as how to overcome low service availability. This challenge emanates from the dynamic nature of the Grid environment and the complexity of services. These two cause services to be fault-prone. Therefore, there is a need to develop an autonomic fault recovery mechanism that will effectively monitor, diagnose and recover a running service from failure.

In addressing the above mentioned challenge, a dynamic service recovery model has been proposed. The model uses the replication approach to improve service availability whenever service failure is envisaged. The performance of the replication approach depends on how well a reliability index can be used to dynamically select two services of high reliability to serve an incoming service request. The service with higher reliability between the two selected services becomes the primary service while the other one becomes an active replica. An autonomic computing MAPE loop is implemented by the model to achieve runtime fault recovery.

A simulation was carried out to evaluate the performance of the proposed model. The model was also compared with the existing active replication model. The results revealed that the newly proposed model exhibits superior performance characteristics especially when there are services with high and low reliability. It was also found that dynamic service recovery efficiently utilizes resource as a result of not-more-than two services serving a request.

CHAPTER ONE
INTRODUCTION

1.1 Overview

Software development and operating environment both change along with the changing computing environment. Software that used to be executed only in closed and independent environments is now executed in a distributed computing environment. Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed computing capabilities. SOA is an information technology approach that emphasizes implementation of components as services that can be discovered and used by clients. By promoting reuse of software components, SOA delivers flexibility, interoperability and cost saving (Gadgil *et al*, 2007). This architecture uses the web service technology to enable distributed resources to be utilized in the current web environment.

The web service technology facilitates the development of software (i.e. web applications) by allowing the integration of independently published web services as components of a new business solution. Lee *et al* (2005) define a web service as a service module that enables a user to receive the desired service through the Internet regardless of time, place, and platform. According to these authors, web services have three enabling technologies. The first one is *Simple Object Access Protocol* (SOAP) that allows the exchange of messages between web services. SOAP is used during service communication over either *HTTP* or *HTTPS*. The second is *Web Service Description Language* (WSDL) that exposes the operation or the functionality of the web service to the service client.

It can be just an ordinary standalone application or any other web service. The third enabling technology is the *Universal Description Discovery and Integration (UDDI)*. This is a registry or repository, which enables service providers to register their web services, and clients to search for web services that will help them to accomplish their business objectives.

Web services are of two types, namely: *Functional web services* and *Autonomic web service* (Guinea and Ghezzi, 2005). Functional web services provide computational functionalities over the Internet. Autonomic web services are services that encapsulate autonomic attribute to provide autonomic behavior over the Internet. Autonomic attributes include characteristics like *self-healing, self-configuring,* and *self-optimization* (Zeid and Gurguise, 2005).

Our main interest in this research work is on self-healing. Ghosh *et al* (2006) define self-healing as the property that enables a service to perceive that it is not operating correctly and, without human intervention, make the necessary adjustment to restore itself to normality. Self-healing consists of autonomic fault detection and recovery from failure to achieve service availability. Ghosh *et al* (2006) further define fault tolerant computing, as the ability of a system to respond seamlessly or with minimal disruption in the presence of fault. Thus, fault-tolerant computing forms part of self-healing.

When a service client requests a service that conforms to a certain task, he or she expects that the required objectives will be fulfilled. This, therefore, requires that web service provisioning must be reliable and fault free. The main challenge is to develop a fault free service while predicting possible failure states for complex web services is even more difficult.

A failure affects availability, reliability and usability. These can be used as measures of Quality of Service (QoS) for web services. Availability and reliability of web services are not guaranteed since web services are stateless from the fact that they don't keep the state of the service. Grid technology extends web services into Grid services by providing some functionality into web services that will maintain the state of the service.

Grid is a dynamic environment where resources are virtualized and shared as if a single machine offers all the resources which, in reality, are geographically distributed and may possibly be managed by different organizations. Given the dynamic nature of the Grid environment, as new resources and machines join the Grid, the need for autonomic fault management that will keep the QoS stable becomes even more important. Autonomic failure recovery in Grid services is required to provide high availability and reliability of the service to the users. The importance of this autonomic behavior is amplified by the fact that such behavior will increase the trustworthiness and QoS of these services to users.

Our research specifically aimed to enhance the *Grid-based Utility Infrastructure for SMME Enabled Technology* (GUISET) (Adigun et al, 2006) to achieve high service availability by automatically detecting anomalies and reconfiguring a system without disturbing service client's task. GUISET is an infrastructure used to support business processes for Small, Medium and Micro Enterprises (SMME) through shared Grid services. GUISET extends Grid middleware to facilitate sharing and management of resources available in a GUISET Grid. GUISET needs a layer of autonomic services in order to guarantee autonomic response to failure. GUISET is the mediator between the service clients and service providers.

Since the GUISET infrastructure supports marketing and selling of SMME products there is a high demand for Quality of Service (QoS). In this work, QoS is being achieved through self-healing.

1.2 Background

SOA uses web service technology to achieve business collaboration. However, this paradigm is still inhibited by quite a number of open challenges. Fault management (Hanemann *et al*, 2004) is one of the areas which still presents some open problems, some of which this work will address. Many strategies have been proposed to address issues related to fault management in Grid services.

Self healing (Fugini and Musi, 2006) is one of the most adopted autonomic fault management strategies for fault handling (Cook *et al*, (2007); (Guinea and Ghezzi, 2005); (Pereira *et al*, 2006). This is because it reduces service down time during fault recovery. This strategy automates fault detection and fault recovery in order to increase service reliability and availability. This research focuses on both fault detection and fault recovery strategies.

What mechanism should fault detection use in order to efficiently detect and report fault occurrence? Though this might sound trivial, in reality it is not easy to enumerate all possible failures in a large and complex system during service execution.

To counter this glitch, Arshad *et al* (2004) proposed a planning-based approach to enumerate occurred fault in order to recover from them. On the other hand, a number of fault-recovery strategies (e.g. (Guinea and Ghezzi, 2005); (Fugini and Musi, 2005)) have been proposed to recover from occurred faults.

Unfortunately, the existing fault-recovery strategies are heavily inhibited by lack of efficiency and transparency during service execution. The inefficiency of these strategies emanates from the fact that there is no standard way of using these fault recovery strategies. Transparency is still a challenge because the more a number of strategies fail in sequence, the greater the response delay. This research investigates fault management mechanisms.

1.3 The Research Problem

Grid services focused mostly on providing computing functionalities without taking into consideration service failures in the process of supplying these services. An optimal service selection provided by the Grid middleware does not guarantee fault free service execution. The dynamic nature of the Grid environment increases the demand for fault detection and recovery during service execution. Availability of accurate and efficient mechanisms for fault detection that take into consideration service provider's QoS expectations for the service, is still a challenge in Grid services. The presence of an accurate and efficient fault detection mechanism will surely improve service recovery. It should also reduce service downtime during the process of service recovery. This research therefore, proposed an efficient, transparent and autonomic fault detection and recovery mechanism in GUISET. This dynamic fault recovery mechanism was crafted with the goal of increasing QoS to the service consumers.

1.4 Rationale of the study

This research work contributes to GUISET and also to Grid middleware especially Global Toolkit 4 (GT4). GUISET is a Grid based infrastructure to enable on-demand services provision to SMMEs.

Due to the dynamic nature and heterogeneity of Grid environment, fault monitoring, detection, diagnosis and recovery are still challenges. GT4 tries to overcome some of these challenges through WS_Reliability and Optimal service selection. WS_Reliabilty guarantees message delivery and elimination of message duplicates. However, this does not provide a mechanism for automatic recovery from service faults during service execution. We hope that our work will ensure service execution continuity when faults occur. This would enhance service trustworthiness, availability and reliability during service execution.

1.5 Research Questions

An investigation of the existing approaches identified the following issues which this research addresses:

1. How can we reduce the occurrences of faults during service execution?
2. How can we reduce mean time to recovery after the occurrence of service failure?

1.6 Goal and Objectives

1.6.1 Goal

The goal of this research was to develop an autonomic service recovery mechanism for Grid services.

1.6.2 Objectives

In fulfilling the goal of this research the following list of objectives will be taken into consideration:

1. To formulate a dynamic fault detection and recovery architecture for Grid services.
2. To simulate the proposed architecture.

3. To evaluate the performance of the simulated architecture.

1.7 Research Methodology

In fulfilling the goal of this research, literature review, model formulation and proof of concept were carried out.

1.7.1 Literature Review.

A literature survey on existing fault detection and recovery models in distributed computing was conducted. The aim was to investigate existing frameworks and standards for fault detection and recovery that relate to this work. The survey for efficient modeling frameworks was conducted together with the analysis of those frameworks with the aim of establishing sound basics for fault recovery mechanisms.

1.7.2 Model formulation

Relevant existing research results were analyzed with the aim of identifying the strong points of existing models. These strengths assisted in fashioning out the design criteria that drove the design of a new model fulfilling our GUISET purpose.

1.7.3 Proof of concept

The proposed model was simulated using J2EE, taking into consideration the Grid environment behavior. The performance evaluation of the model and the experimentation and evaluation of how the model fits the GUISET architecture was then carried out.

1.8 Organization of the Dissertation

The rest of the dissertation is organized as follows:

Chapter two presents the background concepts of this research work. It also lays the foundation for our proposed model. Chapter three presents literature relevant to this study. This chapter starts with the introduction on fault recovery and detection on Grid services. We discuss existing related work, outlining the challenges associated with the design of autonomic service recovery model. The chapter concludes with a brief overview of the proposed model.

In chapter four we present the description of the model development. This chapter begins with an introduction followed by design requirements and the solution approach to solve the problem. The Dynamic Service Recovery Model is then presented with full detail. Performance analysis and a discussion of the result of the model follow. A summary of this chapter is then presented.

Chapter five gives the description of the design of the Dynamic Service Recovery Model with the implementation. The experiments, analysis, performance evaluation of the model and result are presented in this chapter.

Chapter six concludes the dissertation. The recommendations for future work are also presented.

CHAPTER TWO

BACKGROUND

2.1 Introduction

Each company customizes its computer services according to its specific requirements. This has given rise to services which are isolated. On the other hand business growth requires sharing both information and applications which are already available. The emergence of the Internet and the service oriented computing paradigm provides enabling technologies to fulfill this need. In chapter one, we indicated that the goal of this research is to develop a service-recovery mechanism for Grid services. In this chapter we introduce the following background concepts which are fundamental to our research: In section 2.2 we briefly discuss Service Oriented Architecture and Web Services. This is followed by section 2.3 where we discuss Grid Computing and GUISET. Section 2.4 and section 2.5 cover service interaction in the Grid environment and Autonomic computing. Finally we give a conclusion of this chapter in section 2.6.

2.2 Service Oriented Architecture and Web Services

Service Oriented Architecture (SOA) (Dai and Wang, 2006) can be defined as an architecture that separates functions into distinct units called services, which are made accessible over a network in order that they can be combined and reused in the production of business applications. Web services are a solution to the heterogeneity and platform dependence problem of distributed computing. Web service technology provides a uniform framework to increase cross-language and cross-platform interoperability for distributed computing and resource sharing over the Internet. SOA enables services to be published and discovered by service clients.

The common communication protocol in a web services enabled environment is SOAP over Hypertext Text Transfer Protocol (SOAP/HTTP). Web Services Description Language (WSDL) is a specification to describe networked XML-based services and how to access them. It provides a simple way for service providers to describe the format of requests to their systems regardless of the underlying protocols. The service interface (WSDL) is registered to the UDDI registry for service client to discover and match the service capability. After a service client has found a required service, the UDDI successful search allows communication to the provider of the web service. The diagram in Figure 2.1 depicts the interaction between a service client and a service provider.

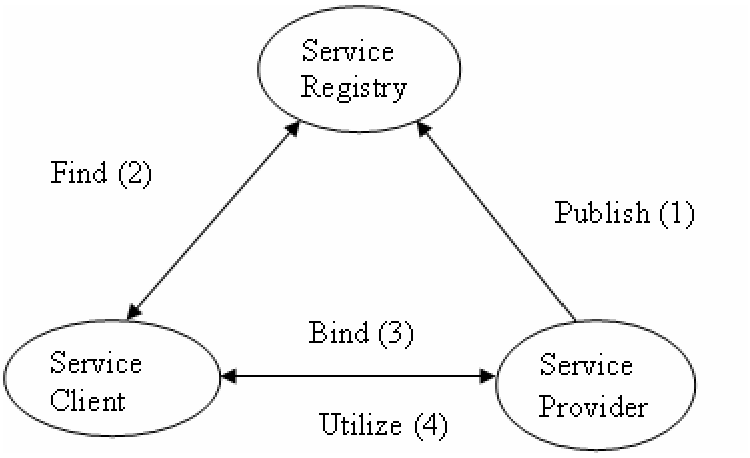


Figure 2. 1 Service-request interaction [IBM, 2004]

WSDL and SOAP technologies rely on XML to make possible interoperability in web services.

2.3 Grid Computing and GUISET

Grid computing is a distributed computing environment where disparate resources such as computer CPUs, storage, applications and data, often spread across different physical locations and administrative domains, are utilized through virtualization and collective management. Figure 2.2 shows a typical Grid computing environment. Grid nodes communicate with one other while Grid access nodes (workstations, laptops, etc) are connected to a particular Grid node. Services in Grid computing are called Grid services. Grid services can be web services with additional functionality to keep the service state. Grid computing uses SOA approach for service provisioning and utilization. Grid computing enables service clients to use services without knowing where those services reside through a Grid middleware.

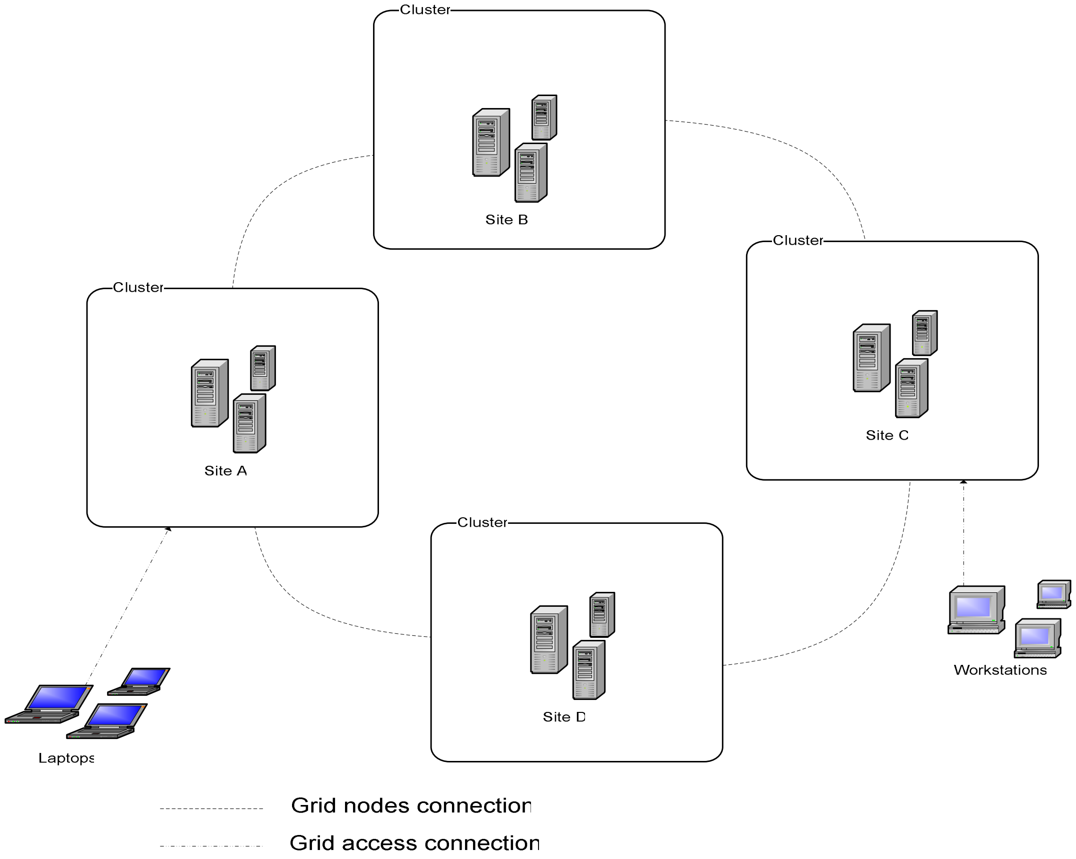


Figure 2. 2 A Grid computing environment showing four sites

Grid middleware (Globus Toolkit, Condor, etc) acts as a broker to achieve service virtualization in each server in a Grid computing environment. Globus toolkit 4 is the reference framework in this research work. Grid middleware allows services to join and leave the Grid network at any time. The Resource Management System (RMS), which is also called the "brain" of the Grid middleware, manages resource matching and execution. All the Grid nodes have a Resource Manager (RM) that manages local execution in each Grid node.

When service clients initiate a request to a particular service, or to access remote resources in the grid, they would send their requests to the RM first because they do not know where the service or resources are provided. The RM will then match the service request with the optimal service and manage the execution of the service request if the service is provided locally. Figure 2.3 shows the Grid RMS and RM interaction. In the process of serving a remote service request, RM interacts with RMS to virtualize the service in the Grid network. The RMS then sends the request to RM where that particular service is deployed.

GUISET has been proposed as an enabling platform for the SMMEs to access ICT services without owning the infrastructure on which the services are deployed. The GUISET services are distributed and GUISET Grid middleware allows the virtualization and access to the services. This concept makes life simpler in service provision for the service provider by taking service provider's responsibility through enabling services usage by the service clients. In this context users (providers, SMMEs and customers) with different capabilities are able to use services without owning the infrastructure and knowing the service provider. Service providers deploy services. SMMEs subscribe for services that help them in their business. Customers also use these services to purchase SMME's products.

This increases GUISET challenges to ensure service availability during service execution to improve service trustworthiness. RM enables GUISET to achieve node by node management of services, while RMS allows service sharing from all the nodes.

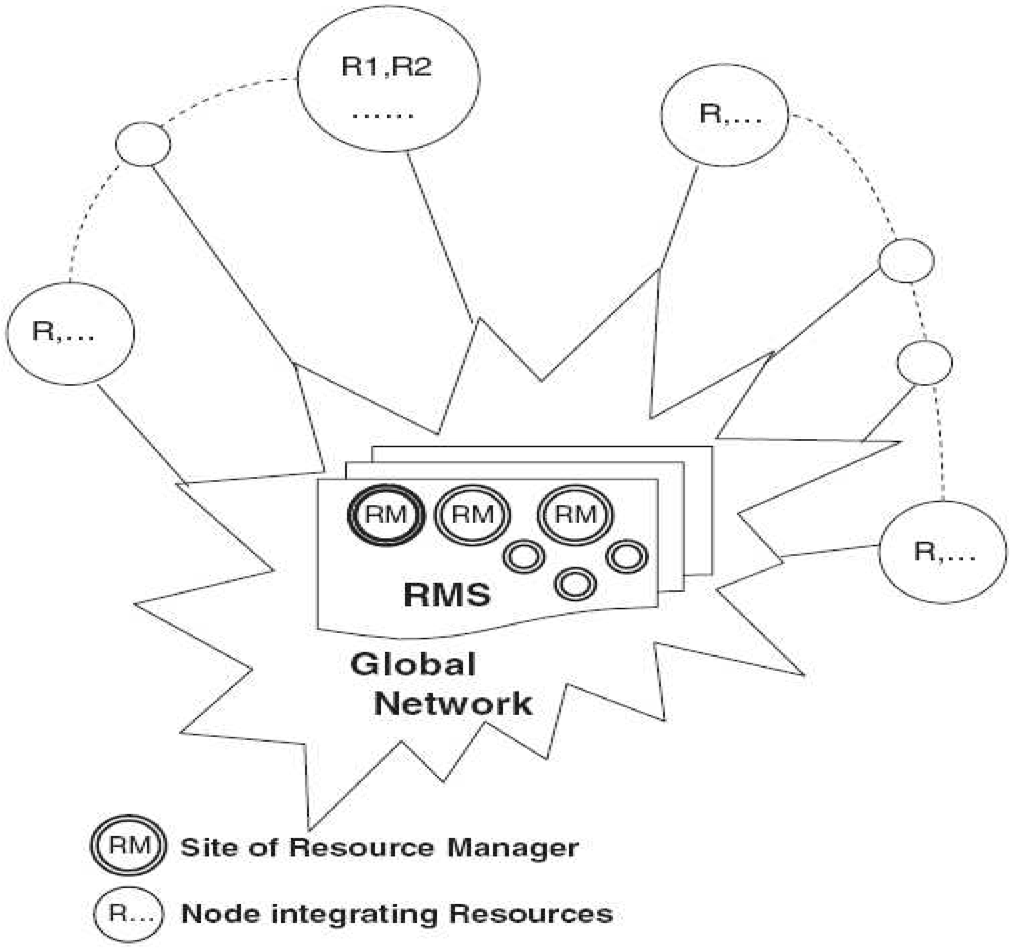


Figure 2. 3 Grid service virtualization [Dai and Wang, 2006]

GUISET Grid nodes are managed by different organizations, which mean that the nodes are running different applications, different processing capabilities, different firewalls and security. During the execution of a particular request, one may find that two or more nodes are required to fulfill a request. This also becomes a GUISET challenge to make sure that user expectations are met for example, service reliability and availability.

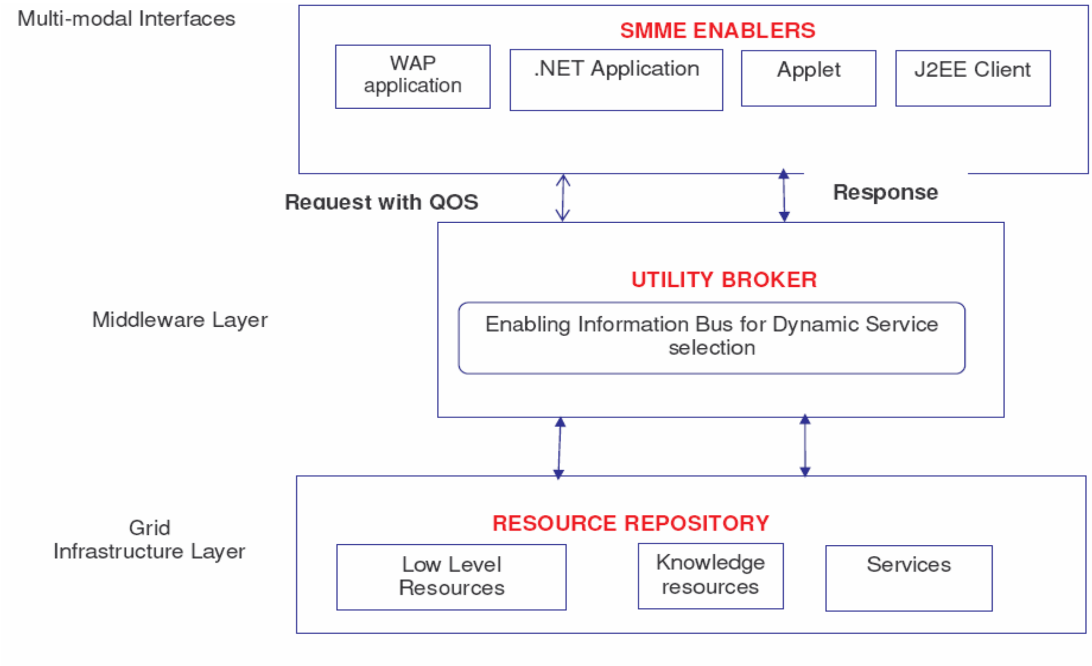


Figure 2. 4 Grid-Based Utility Infrastructure for SMME-Enabling Technologies [Adigun et al, 2006]

A service is available only if service consumers get the service whenever a request for the service is issued. The failure-free web service has to be available until **the** task of the requester is accomplished. Reliability connotes that the web service meets the expectations of the consumer exactly the way the user expects it to be. This also depends on the availability of the web service. Service availability tends to vary dynamically and is dependent upon the service provider's requirement and the environment where the service is running.

Reliable service provides QoS and trustworthiness to service consumers. Usability has to do with how difficult it is to use the service. Reliability and availability are of interest to this research work. Figure 2.4 show the layers of GUISET architecture. The multi-modal interfaces layer of GUISET deals with rendering information to the user while the middleware layer concerns managing the sharing of resources and selection.

The Grid infrastructure layer is where resources and services are hosted. This research is an attempt to address one of the challenges of the middleware layer which is service failure.

2.4 Service Dependability

Grid services reduce both the development work and maintenance work of software development due to their reusability. On the other hand, it raises the demand for self-recovery in grid services during service interaction. In a Grid environment, the larger the number of services that depend on one another, the higher the likelihood of fault occurring. Also the larger the number of individual services involved in a composite service, the greater the possibility of failure of the composite service. Service composition increases service complexity due to service dependability. The diagram in Figure 2.5 illustrates a scenario for requests for possible dependable services.

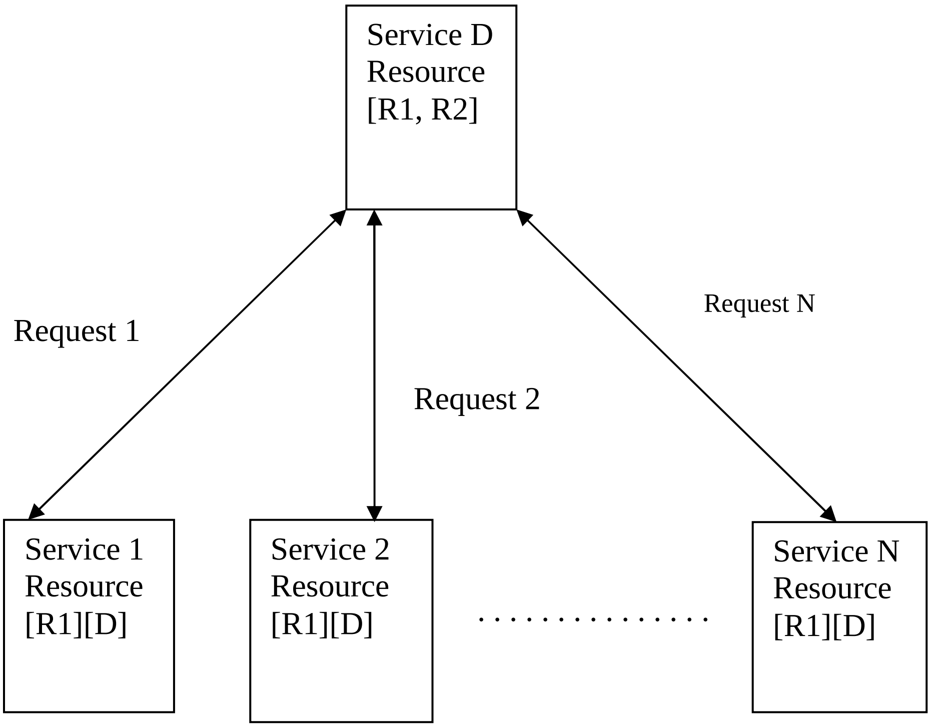


Figure 2. 5 Service dependability

Different services might use the same resource (e.g. service, processor, secondary storage, etc). For example, as illustrated in Figure 2.5, in order to fulfill a particular request, service 1 has to invoke service D which requires the same resource as services 2, 3, 4 up to N. However, since service D is servicing each request from services 1, 2, 3… N, the QoS of service D will degrade because resource R1 is being shared by all the services. This would mean that service D would not be able to fulfill its allocated role in meeting the QoS specified in SLA. The QoS of services 1, 2, 3 … N, depend on QoS level of Service D. Therefore, all of these services QoS would be degraded.

We now take a closer look at composite services. The availability of a composite service in a Grid environment depends on the availability of the other services from which it is composed. The diagram in Figure 2.6 illustrates service composition.

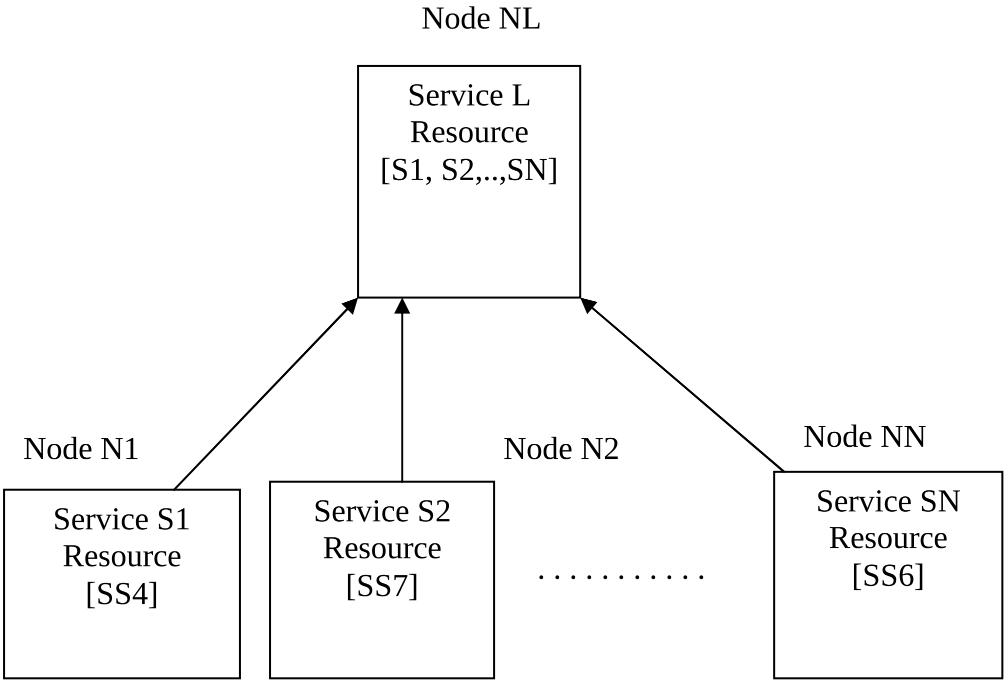


Figure 2. 6 Service Composition

Due to many possible faults such as server crashes, communication failure, etc it becomes more difficult for a composite service such as service L to constantly maintain its availability. The probability that a composite service will fail decreases with the number of services composing such a service. The equation 2.1 depicts the probability of fault occurrence for service L.

$$P(x) = \sum_{i=1}^{n} (P_i) \tag{2.1}$$

Let the probability of service L having a fault when invoked be $p_{i.}$ The total number of node is n. Equation 2.1 illustrates that the more the number of service composing a composite service the greater the probability of fault in a composite service. This raises challenges for service recovery.

2.5 Autonomic computing

The growing complexity of the Grid service platform and their dynamic varying workloads make manual management of Grid service platform a very challenging and time consuming task. Grid service platform is enhanced by autonomic computing capabilities in providing QoS. Autonomic computing has been inspired by human autonomic nervous system (Parasha and Hariri, 2005). Its goal is to realize the way human nervous system works and apply the same behavior in software system. Autonomic computing is a self managing computing model (Kephart and Chess, 2003).

An autonomic computing system must have a mechanism whereby changes in the system can trigger changes in the behavior of the computing system such that the system is brought back into its normal operational state. Autonomic computing has become the most popular paradigm for the provision of QoS in software development and management (Tian *et al*, 2005). IBM defined four components that enable autonomic computing systems to be self managed: self healing, self optimization, self configuration and self protection.

Self-configuration is the capability that enables the system to adapt to unpredictable anomalies by automatically changing its configuration, such as adding or removing new services or resources, or installing changes without disrupting the service.

Self-healing is the capability that the service can prevent and recover from failure by automatically discovering, diagnosing and recovering from anomalies that might cause service disruptions with minimal performance degradation.

Self-optimization is the capability that enables the system to continuously tune itself proactively to improve on existing processes and reactively in response to environmental conditions.

Self-protection is the capability that the service can detect, identify, and defend against viruses, unauthorized access, and denial-of-service attacks. Self-protection also could include the ability for the system to protect itself from physical harm, such as the motion detection capabilities of today's laptops that can temporarily park their disk drive heads if they sense that they are being dropped.

According to the Autonomic Computing paradigm, each self managed system element must be able to: Monitor, Analyse, Plan and Execute (Gurguis and Zeid, 2005); (Kephart and Chess, 2003)).

Each of the four autonomic computing elements implements the MAPE cycle. Our focus in this research is on self healing. Figure 2.7 show the MAPE cycle architecture.

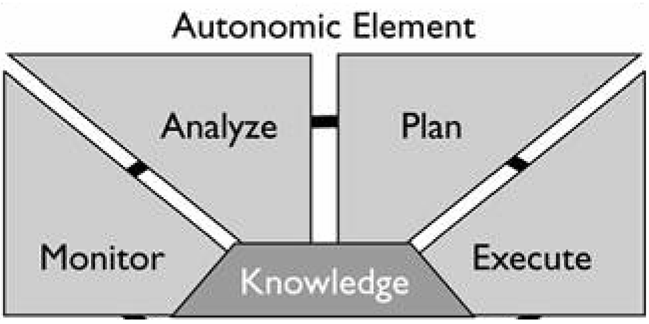


Figure 2. 7 MAPE loop [IBM, 2004]

The monitor element collects, and filters information from the managed service. The analyze element compares these artifacts against a symptom in the knowledge base. The analyze element outputs an indication of any problematic patterns found and a set of possible solutions. The plan element, based on policy data, selects one of the solutions. The execute element carries out the actions for that solution. A central knowledge base, accessible by the other components, contains knowledge pertaining to the likely effectiveness of various possible management decisions in achieving the manager's overall policy objectives.

2.6 Summary

This chapter has laid the background for this research work. GUISET uses the Grid middleware to allow the sharing of distributed services especially enterprise services. The Grid middleware is enhanced by autonomic computing elements to facilitate self management. This research uses self-healing to auto-recover service faults during service execution. The result of this work will also contribute to the Grid middleware as it is also used in the middleware layer of GUISET. Self healing approaches that relate to this study would be discussed in chapter three and the proposed model in chapter four.

CHAPTER THREE
LITERATURE REVIEW

3.1 Overview

An increase in the usage of Grid services in enterprise environment has increased the demand for high service availability during service provision. This challenge raises the need to explore and understand the significance of self-healing systems in engineering the self-management of large-scale complex IT systems. These systems can be comprised of communication infrastructures and computing applications to ensure Grid service recovery during service failure. Self-healing of Grid services should address the concerns of Grid service clients with respect to service availability issues in a transparent manner.

This chapter reviews literature that inspired and supports this research. The first part of this chapter gives a review of self-healing, and it continues with fault detection and recovery. Then we review some related literatures on service recovery in Grid middleware. Finally, we give a conclusion drawn from the reviewed literatures.

3.2 Self-Healing

Self-healing has been used in different areas (robotics, databases, etc) for different purposes, for example adaptation, fault management, etc. According to Mikic-Rakic *et al*, (2002), self-healing system has to exhibit the ability to adapt at runtime to handle situations such as resource variability, changing user needs and system faults.

A Self-healing service is a service that auto-detects abnormal behavior and tries to recover from them. This reduces time and cost of maintaining services, and the skills expected from the person maintaining the service. Self-healing services has to be able to identify faults (fault detection) and to support decision-making to recover (fault recovery) from occurred faults. Self-healing goes beyond detecting and recovering from faults in a sense that it also provides the service with the intelligent capability to change the system state and to report occurred faults by itself or with the assistance from other services.

Self-healing mechanisms can be viewed as a set of autonomic recovery actions fired at run time according to detected faults (Modafferi *et al*, 2006). Ghosh *et al*, (2007) proposed two states which a self-healing system can be in during the process of service provision. The two states are: *healthy* and *unhealthy*. A healthy state is a normative state where the system behaves according to clients and providers expectations. An unhealthy state is an abnormal state where a system does not behave according to the user's expectation. A self-healing system needs to know what constitutes healthy and unhealthy state before it can make any adjustment to restore itself to a healthy state.

Self-healing according to Ghosh *et al*, (2006) and Ahmed *et al*, (2007) can be subdivided into two components that maintain system health through system monitoring:

a. Detection of system failure that deals with fault detection, and

b. System recovery that deals with recovering from failure.

Detection and recovery processes must not cause degradation of running services. This factor needs to be considered during development self-healing components.

Shin and Hoon, (2006) proposed component based self-healing where each software component has a service layer and a healing layer. A service layer provides the functionality of the component to other components, while the healing layer provides a healing mechanism to the component. This kind of self-healing can work well in static or simple environment. In a GUISET environment this type of self-healing mechanism cannot work because it will degrade service performance because it is not automated and it has a lot of stages. Figure 3.1 shows the healing layer process as defined by Shin (2006).

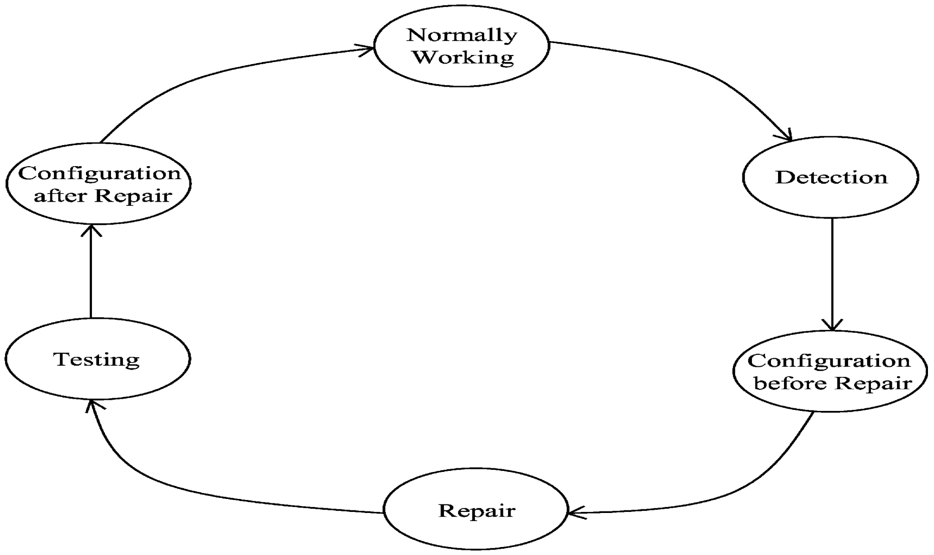


Figure 3. 1 Steps in the self-healing process (Shin, 2006)

Self-healing services have to reduce fault service downtime and be efficient without degrading the performance of other services. Different requirements of self-healing systems have been proposed to overcome the above challenge. Net and Muller (2007) suggested that traditional quality attributes of self-healing (reliability, availability, etc) are not enough in providing QoS as far as self-healing is concerned. The additional quality attributes proposed by Net and Muller (2007) are: dynamic adaptation support, dynamic upgrade support, diagnostic support, and support for accountability. The above quality attributes can work well in some of self-healing areas for example, self-healing for adaptation.

In the context of service recovery, some of these quality attributes like dynamic adaptation support will cause overhead. Different mechanism and approaches have been used for fault detection and recovery, some of which will be considered in the next sections. Fault detection plays a major role for system management and other components like fault prediction.

3.3 Fault detection

A considerable amount of research has been carried out in the field of fault detection. Due to the increasing use of services, the issue of fault management has taken centre stage. Fault detection that will increase trustworthiness from the service clients and improve service availability through autonomic and accurate detection is needed.

A lot of fault detection approaches have been proposed (e.g. (Berharref *et al*, 2005), (Pereira *et al*, 2006) and (Modafferi *et al*, 2006)). But only two are of particular interest to this study: fault detection architectures proposed by Berharref *et al* (2005) and self-healing middleware proposed by Pereira *et al* (2006). Fault detection architectures use passive testing for faults in web services. The fault detection architecture proposed by Berharref *et al* (2005) uses an observer web service that can be invoked either by the requestor or by provider of the service.

This architecture can work in environments where both the service requestor and the service provider do not have any quality of service agreement. This architecture cannot be used for autonomic web services since there is no transparency in the process of handling fault, because the client can see that an error occurred.

This architecture increases human intervention when it is handling fault. In the self-healing middleware (Pereira *et al*, 2006), two fault detection mechanisms have been proposed, pre-emptive and on-use fault detection. Pre-emptive fault detection is used on a regular basis, while On-use fault detection is only used during service invocation.

It is important that self-healing systems have strong fault detection abilities to increase service reliability, because the reliability of other components (e.g. fault recovery) depend on it. The pre-emptive mechanism is the most widely used fault detection mechanism, whereby the healing service periodically checks for fault occurrences. On-use detection can work well in environments that are not fault prone. On-use detection cannot be used in a dynamic and complex environment like Grid, because it will cause service recovery overhead and it will increase service downtime. The self-healing middleware does not take user objectives into consideration because it does not use users objectives during fault detect.

According to Dabrowski *et al* (2003), applications using discovery systems rely on a combination of two techniques to detect failures. The two techniques are: monitoring periodic transmissions, and the retry ad-hoc transmission. System components listen for recurring messages, such as heartbeat messages, and failure to receive such messages will mean the component has failed. These techniques can work well in environments where service demand is low and environments that are neither complex and nor dynamic. In environments like Grid, it is difficult to use these techniques since service availability is dynamic. And it is difficult to predict service demand. It is also difficult to figure out the root cause of the failure. Network overhead will increase because of the heartbeat messages that will be sent by related services.

Duan *et al* (2006) proposed a data mining based fault detection and prediction service. Fault detection and prediction depend on data that is captured. Fault prediction based on data mining can also work for Grid environments only if one uses service's group data for prediction. Fault detection based on data mining has some limitation in the sense that in Grid environments there is no guarantee that service specifications will not change.

In the research conducted by Baresi *et al* (2005) two types of runtime error discovery were proposed:

1. Defensive Process Design and
2. Service run time Monitoring.

Defensive Process Design deals with designing the service oriented business process in such a way as to permit it to cope with erroneous behavior. Service run time Monitoring uses an external monitor service capable of checking whether functional and non functional contracts are not violated. The interest of our research is using the external monitor to monitor abnormal behavior in services.

In considering developers expectation in the process of fault handling, Tang *et al* (2005) proposed an approach for fault handling. In this approach, fault specifications and the corresponding handling mechanisms of the services are both defined in service policies. Each service has its own policy on fault specification and handling. The service policy is managed through out the execution of the service. This approach uses periodical monitoring on the running service. Figure 3.2 shows the components and their interaction during service provision.

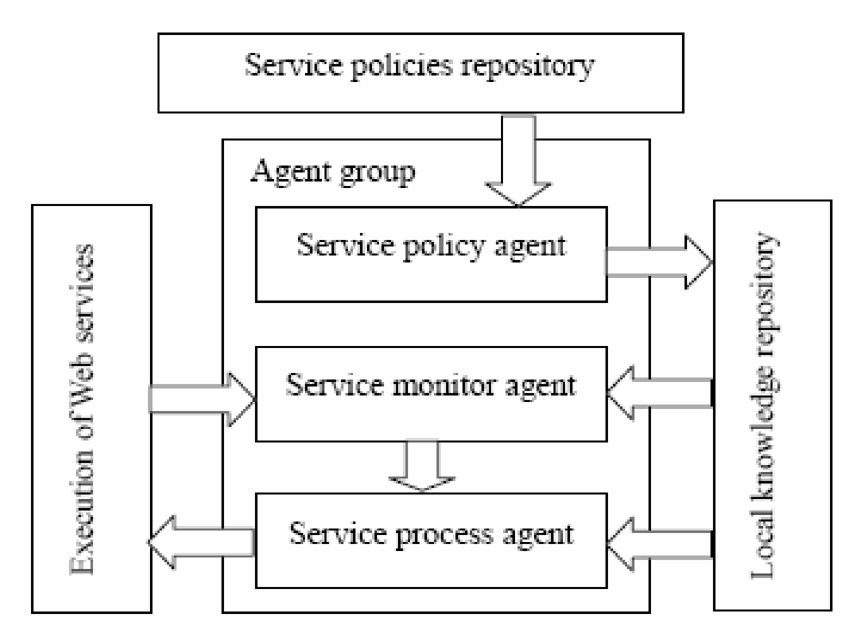

Service policies repository
Agent group
Service policy agent
Service monitor agent
Service process agent
Execution of Web services
Local knowledge repository

Figure 3. 2 Policy driven fault detection (Tang et al, 2005)

In summary, fault detection plays an important role in a self-healing service in a way that the other components of self-healing cannot function without it. Fault detection triggers other components (such as fault recovery) when a fault is detected. In that way, fault detection becomes the backbone for fault recovery since the service cannot do anything if there is no fault detected.

3.4 Fault Recovery

Fault recovery has a long history in distributed relational database and distributed systems. Fault recovery is the most challenging part of self-healing since it has to support decision making. The rapid growth of the use of services and the dynamic nature of the Grid environment require some intelligence in the process of decision making. In this section we discuss different recovery strategies that have been proposed in the literature on self healing that relates to this research work. Various approaches have been proposed to recover from software faults: fault prevention, fault removal, and fault tolerance (Jiang *et al*, 2007).

Fault prevention aims to achieve fault free software through robust design and testing. Fault removal aims at transparent recovery from faults. Fault tolerance aims to ensure continual operation of a system in the presence of faults. Section 3.4.1, Fault removal, and Section 3.4.2 fault tolerance and prevention, cover related work under the afore-mentioned fault recovery approaches.

3.4.1 Fault Removal

A retry recovery strategy has become the most used recovery technique for transient non deterministic fault (Fugini and Mussi, 2005). A transient fault does not need any change either in the environment or in the service in order to recover. Memory overflow is one example of transient faults. Recovery from such faults needs re-invocation of the service.

Arshad *et al* (2004) proposed the planning based approach for fault removal recovery. This approach automates failure recovery by capturing the state before the failure. The initial state and goal state need to be supplied to the planner to plan for the next action. This planning based approach has been tried in distributed systems. According to Arshad *et al* (2004) the use of AI planning in distributed systems is one of the recovery techniques that have the ability to minimize time, cost and resource usage.

3.4.2 Fault Tolerance and Prevention

Since a Grid service is a software application, other traditional recovery mechanisms to tolerate faults can be encapsulated during application development. For example, exception catching mechanism is one of the traditional software recovery strategies.

These mechanisms are used for trapping runtime faults without finding the source of the faults. They also increase performance degradation since they do not resolve faults. Designing a fault tolerant Grid service has become a challenge. Martinello *et al* (2005) identified two types of error recovery strategies: Non client transparent and Client transparent. Non client transparency does not provide transparent request handling during node failure occurrences.

Client transparency enables web requests to be smoothly migrated and recovered on other working nodes in the presence of node failure in a user transparent way. A client transparent approach can work well in a Grid environment because of the high demand of service availability and because of the contract that binds both the service provider and the client. However, this approach does not consider service execution continuation from the fact that the working node has to start from the beginning of the execution process. This research uses a client transparent approach for fault recovery.

A client transparent approach can be achieved in two ways (Tartanoglu *et al*, 2003): backward (based on rolling system component to the previous correct state) and forward fault recovery. Backward error recovery increases response delay due to the fact that even if a service was about to finish then everything will be rolled back. Forward fault recovery involves transforming the system component into a correct state. Forward error recovery is of interest to this research since it optimizes service execution time.

3.4.2.1 Replication based fault recovery

Replicating web services can offer client applications a number of QoS benefits, including higher availability and reduced response time, by allowing client requests to invoke a replica that is less loaded. Section 3.4.2.1.1 and 3.4.2.1.2 give details of different types of replication approaches.

3.4.2 .1.1 Replication based on both same or different replicas

Fang *et al* (2007) proposed a FT-SOAP based fault tolerant mechanism composed of four functionalities:

a. Replication manager,

b. Fault manager,

c. Logging or recovery mechanism and

d. Client Fault Tolerant transparency.

Replication management includes group constitution and membership management. When a fault occurs, the recovery mechanism selects a new primary server that acts as a backup. The new primary will perform the recovery process and select a new backup server. This approach cannot work for delay sensitive software applications because each server in the network will have its own workload. This results in incoming workload from the failed primary server to queue for execution on servers with degraded and varying performance levels.

Replication or Redundancy recovery techniques have become the most popular recovery mechanisms for both fault tolerance and fault prevention (Fang *et al*, 2007), (Ghosh *et al*, 2006), (Parashar and Hariri, 2005). In this research work we look at redundancy as divided in three parts:

a. Local redundancy,

b. Remote redundancy and

c. Hybrid redundancy.

Local redundancy means the service has replicas only on the local node. In remote redundancy, the service has replicas only on the remote nodes. Hybrid redundancy implies the service has replicas in the local node and also in the remote nodes. Remote redundancy is most used for hardware faults while the other two are mostly used for software faults. Replication processes can be active or passive (Treaster, 2005). In the passive replication model, only one of the replicas, known as the primary replica, receives and responds to client requests. Gokhale and Dasarathy (2007) further categorize passive replication into two: Warm and Cold.

In warm passive replication approach, one or more backup or secondary replicas are always running and the state of the backup replicas is periodically synchronized with that of the primary. In cold passive replication, the backup replicas are cold, as the name indicates, in the sense that replicas are not running. Only when the primary fails, one of the idle replicas is selected and the state of the failed primary is loaded into that replica, which then becomes the new primary server.

When the active service fails, the replica is required to bring its state up to date from the last synchronized state and continue with the execution. The passive replica discards the synchronized information, if the active service has not failed.

In active replication each coming request is sent to all replicas. For active replication to function correctly, totally ordered reliable group communication needs to be used to deliver requests to all the replicas in the same order. Although duplicate responses would be returned, only one response is forwarded to the client and the others get discarded

Pereira *et al* (2006) propose that for each service failed, the service has to be replaced with the service from the lookup repository where all services are registered. This approach will not solve service independent faults if the failed service and the lookup service run on the same node. The work done by Yoshikawa *et al* (2003) described a platform that realized rapid recovery by switching to service alternatives to guarantee high reliability.

This platform achieved rapid service recovery through failure detection, service discovery and service switching. This rapid service recovery can work well for services provided by the same service provider because different providers have different ways of developing a service. Maintenance cost in using this recovery technique is not considered in the sense that when the service fails it will wait for the service provider to find out that the service is faulty. This approach does not take into consideration service downtime from the fact that the increase in the number of switches will cause the increase in service downtime.

Lee *et al* (2005) proposed the UDDI "tmodel" that is used in order to find and connect to an identical service for backup. It also describes the value set specification to indicate which agreement, specification and standard the service complies with. The tmodel selects a service based on quality measurements. This technique works well for unrecoverable faults, because unrecoverable faults need to replace faulty services.

3.4.2 .1.2 Replication based on the same replicas

The backup approach suggested by Fang *et al* (2006), Jin *et al* (2004) and Ardissono *et al* (2006) extended the work done by Pereira *et al* (2006) by allowing service lookup from the backup vary for each service. The service backup approach only considers system crush faults. This approach guarantees the continuous execution in the sense that if one system fails; the other system will be able to take over. This approach increases service complexity because a replication manager has to be added for the management of backup services. This approach will also increase fault detection challenges, because the capabilities from one Grid node to another may vary, since varying of node's capabilities causes a change in fault detection specification.

In the work done by Maximilien and Sing (2003), a proxy for service selection was proposed that a web service application instantiates each service it plan to use. The proxy selects services based on service reputation. This idea of proxy works well in distributed systems where the environment is not dynamic. The approach will not work in a Grid environment, since service discovery consists of too many processes (Hasselmeyer, 2005) and services join and leaves the network dynamically.

Grid environments allow all the nodes to register their services through the Grid middleware in order for services to be virtualized. This makes the proxy approach not applicable in a Grid environment. Discovery processes consist of service registration and look-up processes. A look-up process done during service development is called a static look-up while the other one at runtime is called dynamic look-up.

Proxy caching will also not help because all requests in a Grid environment are submitted to the global resource manager to be managed and scheduled. Service dependability in Grid will cause the proxy recovery technique not to function since each and every service depends on different resources. Tang *et al* (2005) proposed that a fault tolerance service should implement two approaches for recovery: the Primary Backup approach, and the state-Machine approach.

The Primary Backup approach has to tolerate crash and omission faults. The State-Machine approach has to tolerate faults such as arbitrary or byzantine faults. In this approach, fault specification and the corresponding handling mechanism are both defined in the service policy. This approach can work well for services that are not complex (Arshad *et al*, 2004). But with this approach it is not practical to enumerate all possible types of faults in a complex service.

3.4.2 .2 Checkpointing based fault recovery

Checkpointing recovery is a recovery mechanism whereby the state of the service is saved periodically during its healthy state of operation Treaster (2005). After a fault has occurred, the affected service rebuilds its state from the last healthy state saved and continues with the execution.

Checkpointing recovery has different types: coordinated and uncoordinated. Uncoordinated checkpointing connotes that a service decides when to synchronize its state information, while coordinated checkpointing is controlled. Coordinated checkpointing is of interest to this research for the fact that state information in Grid environment is managed in the middleware. Dabrowski *et al* (2004) proposed two types of recovery techniques based on coordinated checkpointing: the soft state and application level persistence.

The soft state is when an application announcing periodically soft information about its state. Application persistence is achieved by periodically caching the state of the application.

Each time a new announcement is received, the receiver overwrites the previous cached state. When an announcement fails to arrive, a receiver discards the previous cached state, and when announcement resume a receiver rediscover the latest application state. These recovery techniques increase network overhead and also rely on an external component to come out with recovery actions. These recovery techniques can work well for checkpointing.

3.4.2 .3 Multiple recovery strategies based fault recovery

One effort towards the goal of having a recovery strategy in web services is presented in the work of Guinea and Ghezzi (2005). These recovery strategies are: retry, substitute and restructure. These strategies can also work well in environments that are not fault prone, and also for faults that are not complex, for example faults that rise from resource overload. These strategies can also affect the availability of the service and trust from the users because it is not guaranteed that after using one or two strategies, one would come out with a solution. Another effort towards fault recovery by Tang (2006) adopts a mechanism of predefined policy driven fault monitoring and handling. This mechanism was used to monitor and handle faults in the services that are running on different servers. When a fault occurs in the service, this mechanism restarts the service. This mechanism will increase the response delay. Monitoring services in different servers use messages between servers. Increases in the number of messages passed between the servers, results in increased request delay. This mechanism can therefore not be used for transparent fault handling.

The work done by Fugini and Mussi (2006) tried to solve the problem of efficiency in fault management.

Web service execution faults and coordination faults were the two types of fault identified in this work. The research further categorizes recovery action (retry, substitute and restructure) according to fault type. This recovery strategy can also not overcome the issue of fault transparent handling due to the fact that it only concentrate on one fault at a time, for example if the failure solution is to substitute the service, it will only substitute the service without checking if a parameter or method naming matches with that of the failed service.

3.5 Summary

From the fact that different kinds of faults can occur during service execution, this then require a hybrid based fault recovery mechanism that combine retry, hybrid replication and check pointing through Globus Toolkit four. In our self healing service the policy based fault detection approach will be used for fault detection. Service downtime is taken into consideration in our model.

CHAPTER FOUR

MODEL DESIGN AND DEVELOPMENT

4.1 Introduction

This chapter describes the design and prototype development of the proposed dynamic fault-recovery model. The chapter begins by discussing design requirements of the proposed model. The importance of addressing the problem of dynamic service recovery is also highlighted. Thereafter, key requirements for dynamic fault recovery are identified. This is followed by an in-depth discussion of the proposed dynamic fault recovery model. An explanation of some of the design concepts is also presented. The chapter concludes with a discussion of how the model can help overcome service recovery limitations.

4.2 Design Requirements

The Grid environment is an open and dynamic environment characterized by autonomous entities. The implication of this is that such entities are capable of manifesting uncertain behavior. During service interactions, an abnormal behavior from any of the collaborating services can result into failure of the entire system. Grid service failure is the most challenging abnormal behavior for both service providers and service clients (Huda *et al*, 2005; Qian-mu *et al*, 2006). Service providers offer Grid services that expose service characteristics to service clients. The Grid middleware provides an environment where Grid services are virtualized in all Grid nodes without fault management capability. Service clients are only eager to use services to fulfill their objectives. Fault management challenges affect service clients' expectations.

Fault management in Grid environments need to provide a high degree of service availability, reliability and efficiency. In this case, a service client can be either a user through any application or another Grid service. Grid services can be composed to accomplish a particular task. Grid service composition influences service complexity and that can lead to fault prone services.

In order to elicit the design criteria for this work, we considered the following scenario instance of the complexity of interaction in a Grid service environment (Huhns and Singh 2005, page: 76):

> *Let's consider a typical surgery division in a large hospital. The hospital system is composed of an integrated payroll, scheduling, and billing services. Each service is quite complex, with its own operations and databases, perhaps running on different operating systems. For obvious reasons, these services must work together. Scheduling employees and operating rooms for surgery is complicated, for example, because schedules require frequent updating. A scheduling system must balance staff and equipment availability with unpredictable levels of surgical urgency and advance notice. The mechanisms for payroll are similarly complex – the payroll service must consider various kinds of overtime rules for different categories of labor, such as nurses, residents, consulting physicians, senior surgeons, radiologists, and so on and rely to some extent on data from the scheduling system. Likewise, the billing service must also incorporate scheduling information. It is used not only to bill customers, but also to deal with medical insurance companies and government agencies (such as those for children, the elderly, retired government employees, and veterans). Agencies typically impose complex rules for valid billing and penalties for violations of these rules.*

It is clear that the complexity of these services increases the complexity of the hospital system. System complexity increases challenges for system administrator to manage system failures. From the above scenario we can outline some basic requirements for services during interaction:

a) **Autonomic behavior**: service recovery has to automatically figure out each service abnormal behavior and act on it. Service recovery needs to improve service availability during each service malfunctioning.

b) **Service Trustworthiness**: Service recovery has to increase trust from hospital employees and agencies using the service. Service trustworthiness is important because of the agreement between the hospital and its service providers. Service trustworthiness comes from service reliability, availability, and efficiency that each service has to provide.

c) **Complete fault detection:** Abnormal behavior might occur during the process of each service execution. For service recovery to be triggered, an abnormal behavior needs to be detected from the service. Abnormal behavior characteristics need to be completely outlined. Fault detection also needs to be complete in the sense that all faults that occurred need to be detected. Complete fault detection also helps to reduce fault recurring, for example, fault from scheduler affecting payroll service.

d) **Awareness:** service recovery processes must support the monitoring of each service's state. Various measurements related to service reliability need to be measured. The measurements must be performed and any recovery action needs to be implemented if one of the services underperforms.

e) **Adaptability:** service recovery processes must have the ability to change the system structure, topology and interaction at run time to keep the hospital system up and running in the presence of service failure.

From the foregoing, each of the above defined hospital services can assume any of the four states: *active, inactive, faulty,* or *dormant* as shown in Figure 4.1. In the *active* state, the service is operating as specified in the Service Level Agreement (SLA). In a *faulty* state, a service shows some abnormal behavior. In this state nothing has been done to recover from the fault. After some recovery mechanism has been applied to the service, then the service might be in the *active state* if it manages to recover, otherwise it will be in the *dormant* state. Dormant mean there are still requests of the service while inactive state means that there are no service requests to the service.

After an unrecoverable fault has been discovered, the service will be in dormant state. Thereafter, it will be in *inactive* state because no client will be accessing it and also new services join in this state.

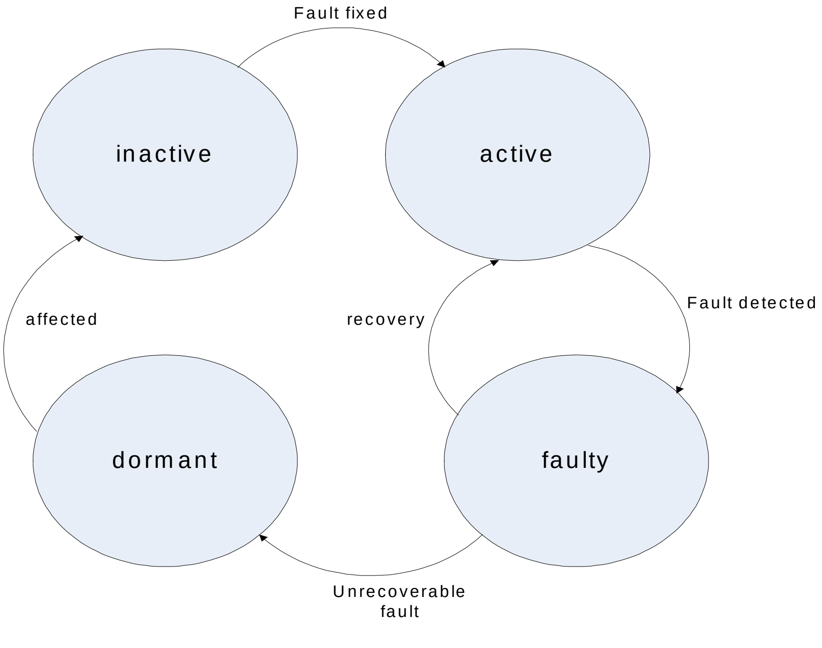


Figure 4. 1 Service transition diagram

There are many possible unpredicted faults that may cause a service to malfunction especially in complex services (Arshad *et al*, 2004). In literature, different fault types have been identified (Chan *et al,* 2007; Bruning *et al*, 2007).

Our research has identified three categories or classes of faults that can occur during service execution from the scenario panted above: **parameter mismatch**, **service overload** and **service unavailability**.

The *Parameter mismatch* fault class is for all faults that occur because of parameter related misbehavior. A wrapper has been proposed in the literature to recover from this category of faults. The *Service overload* fault class is for all faults that occur when the service is in a busy state. The most used recovery action for this category of faults is to block incoming requests.

This fault class is different from the case where the service is not operating at all. In this case, the service is up and running, but the maximum number of services it can serve has been exceeded. *Service unavailability* is a class of faults resulting from services not found. This is when a service exists, but fails to get through due to faults or other request factors. Some of these faults can occur due to the dynamic nature of Grid service availability. There is no recovery action currently in Grid environment to overcome this category of faults.

There are many possible faults that might fall into any of these three classes. For example service interaction may cause a particular service to fail. The following scenario outlines faults that may occur from service interaction:

Service A depends on the output of service B. The provider of service B decides to do some operations on service B. Before service B was passing two parameters, but now it is passing three parameters. Service A invokes service B and passes two parameters as usual.

Service A fails because of no output from service B due to the number of mismatching parameters in service B. This kind of faults falls under the parameter mismatch class. During the process of removing Service B, other services that would have been using service B would also fail due to service unavailability. We have then proposed an integrated fault recovery approach to address these fault categories. Table 4.1 gives the details of the three fault classes identified and their possible courses of fault.

Table 4. 1 Fault classes

Fault class	**Explanation**
Parameter mismatch	• The service client might pass incorrect parameters • The service might not receive any parameters because the service that was supposed to pass parameters failed or has a fault. • The service may receive input that triggers dormant fault in the service for example in the case of division by zero. • The service output does not conform to the input of the service that consumes the output.
Service overload	Since in a Grid may use services that are managed outside the Grid environment and also in different infrastructure, may find that a service is not capable of handling service requests greater than a particular threshold. This may cause a delay or even outright feature of some service requests.
Service unavailability	This type of faults may occur due to the dynamic nature of the Grid environment where services leave and join the Grid network without restrictions. Also this faults may occur because of service malfunctioning that leads to no output from the service.

Table 4.2 outlines some possible fault recovery mechanisms for faults classes identified. The proposed recovery actions are also presented for the identified categories of faults.

Table 4. 2 Possible fault recovery mechanism

Fault class	**Recovery action**
Parameter mismatch	• A service wrapper is used to wrap the output to the way that the depending service client will be able to consume the service. • If the service output cause mismatch of parameters specified in the SLA then the recovery plan has to get the service that is doing the same thing that can produce consumable output. • If the service is not available during its turn the planner has to allow replacement of the failed service by the new service in a group of the failed service.
Service overload	The recovery action here is to find the same service in a service group of the overloaded service to assist the service or prioritize services clients.
Service unavailability	In this case the recovery action is to replace the service by getting a new service provider for the same service in the group of the service that is not found or failed.

4.3 Model Architecture

We propose a Dynamic Service Recovery (DSR) model to address the limitations identified in the literature. The proposed model, by using autonomic behavior when a fault occurs, reduces human intervention during the process of recovering from failures. The autonomic behavior is through self healing where faults are detected automatically, and the recovery process is through the implementation of elements of autonomic computing architecture. The model considers the issue of service availability, reliability and trustworthiness, which is expected when the service client uses the service.

The proposed model will cater for all recoverable faults (i.e. faults that do not need human intervention), but the service provider and service client would be notified of faults that need human attention. This proposed model would also show how it uses knowledge based information to figure out possible solutions for all faults that occurred. This model would also take care of the issue of transparency by reducing delay through allowing the most reliable service to serve the request.

The model would be distributed in all nodes that would join the GUISET Grid network and this distribution would also reduce the response delay during fault recovery. This model would use asynchronous communication. Figure 4.2 shows the architecture of our proposed model for service recovery, the DSR model.

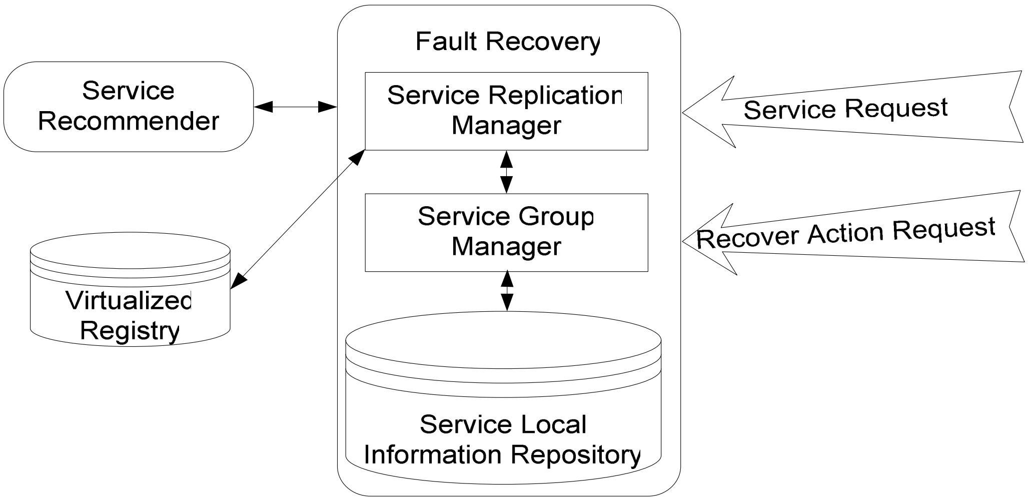


Figure 4. 2 Dynamic Service Recovery Model

The DSR model uses a fault detector proposed by Tang (2005) to address fault detection challenges mentioned in the literature for dynamic service recovery. The fault detector component and fault recovery ordinary components interact during service provisioning. The two components use text messaging to communicate. The fault detector component is employed to achieve complete fault detection and awareness. The fault recovery component is proposed to address service trustworthiness and service adaptability during service provision. Each component has interacting sub- components to accomplish the goal of the main component. In this work, only the fault recovery component is discussed in detail.

4.3.1 Fault Recovery Component

Service recovery has to be able to support decision making on the recovery action that needs to be executed to achieve the normal state of the service.

The proposed model uses some of distributed computing mechanisms to figure out possible solutions for all recoverable faults during service execution. This fault recovery component is composed of three sub-components:

i. Service Group manager
ii. Service replication manager
iii. Service Recommender

4.3.1.1 Service Group manager

The Service Group manager is composed of three modules which are:

i. Service interceptor module,
ii. Service dedicator module and,
iii. Request distributor module.

Service interceptor module receives arriving requests from service clients. It also attaches a request identifier that enhances the request to be synchronized. The request is then passed to the service dedicator and waits for the arrival of a response. When the response arrives, the service interceptor module passes it to the service client. The service dedicator module ranks the services based on their history of usage. This module also updates service history when a service client accesses a service and when a service successfully completes serving a request. The request distributor module multicasts request to recommended services. After multicasting the fault detector takes over to monitor and detect faults.

The service group manager makes sure that each and every fault reported get attention. The request distributor module is triggered when it gets the request to execute. It also depends on the recommender, which has to recommend the services that have to execute a particular request.

The service dedicator module also handles faults that occur during service provisioning. The planner is used to address the process of recovering the failed services.

4.3.1.1.1 Planner

Planning has a long history and it has been used to address a number of problems in Artificial Intelligence (AI). Planning has been used in optimizing search engines and finding optimal solution for AI related problems (Arshad et al, 2004 ; Ghosh et al, 2007). In our model, the planner was used to construct and filter optimal and efficient services to execute an incoming request so that the occurrence of fault could be reduced during service provisioning. A plan is an ordered set of actions that is needed to repair detected abnormal behavior.

At the most basic level, the purpose of the planner is to find a sequence of actions that changes an initial state into a final state that satisfies a goal statement. The model uses the basic level purpose of the planner with additional functionality of filtering the optimal plan from the sequence of actions. The planner undergoes analyzing, generation and selection processes to come up with a fault recovery solution. Analyzing is when the planner processes the information about the domain of the service. Generation is the process when the planner discovers possible plans for a particular fault that occurred. Selection is the process where the planner filters out the optimal plan for a given fault that occurred. Figure 4.3 shows the execution sequence of the plan processes.

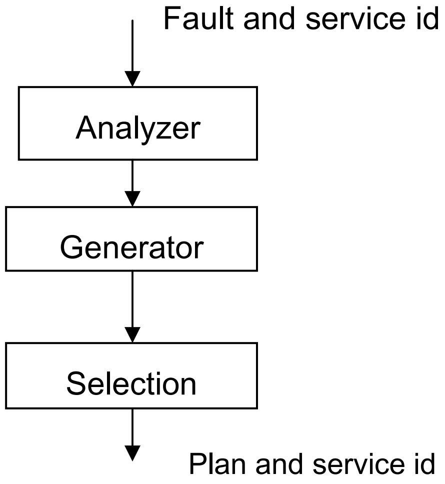
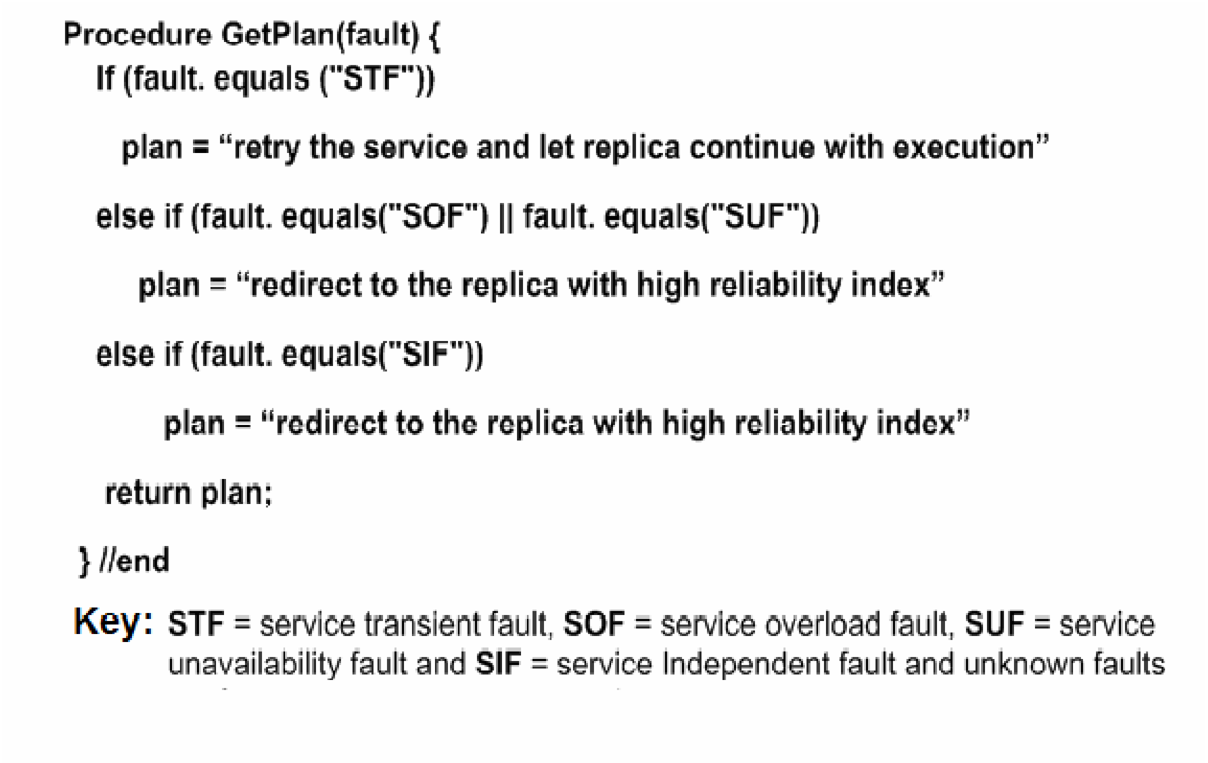


Figure 4. 3 Planning process

In the process of service provisioning and service utilization, new faults and other faults that are similar to fault already identified will be detected. For all new faults detected, the planner uses the service history to figure out the possible plan for the new fault that occurred. For all new faults that needs human intervention the planner reports those faults to the administrator in a way that it will be easier for the administrator to locate where the fault has occurred. Figure 4.4 shows the algorithm used by the planner to come up with a fault recovery solution to find a service with high reliability index, using the service recovery algorithm in Figure 4.5:



```
Procedure GetPlan(fault) {
  If (fault. equals ("STF"))
     plan = "retry the service and let replica continue with execution"
  else if (fault. equals("SOF") || fault. equals("SUF"))
      plan = "redirect to the replica with high reliability index"
  else if (fault. equals("SIF"))
        plan = "redirect to the replica with high reliability index"
   return plan;
} //end
```

Key: **STF** = service transient fault, **SOF** = service overload fault, **SUF** = service unavailability fault and **SIF** = service Independent fault and unknown faults

Figure 4. 4 Planner algorithm

For parameter mismatch faults, the fault recovery component keeps the wrapper plan for future use, since these faults are caused by the service clients. We assume that this kind of faults is caused by service replacement. The DSR model replaces the service and alerts the service provider if no recovery plan found.

4.3.1.1.2 Fault Isolation

Fault isolation is an important process after the process of successful service recovery. Service faults need to be eliminated for future functioning of the service. This reduces service recovery overhead, in a way that if another service invokes the same service, it will not experience the same fault. Fault isolation mechanism varies according to the category of faults, for example faults that fall under the category of parameter mismatch are assumed to be service client based. While the other two categories (service overload, service not found), which are service dependent faults that are from service provider.

In case of service load faults those faults will not be isolated. This is because those faults are caused by the unavailability of resources (memory, processor, etc) that the service depends on, not by the unavailability of the service itself. These kinds of faults are short term faults. Service providers get notified for all faults that occurred in each service.

4.3.1.1.3 Recovery algorithm

We give the description of our dynamic service recovery algorithm, which is well elucidated in Figure 4.5. The algorithm was developed to improve the functionality of the service group manager and service availability during service execution. The algorithm outlines how the modules of a service group manager interact during service provisioning. During the process of accessing the remote service, the remote DSR does not interfere with the service request from the fact that the service is accessed directly from the service registry. The algorithm utilizes the information on the existing services and the usage history of each service to commit a service for an execution, service usage history (number of service accessed, number of service failure, number of service success).

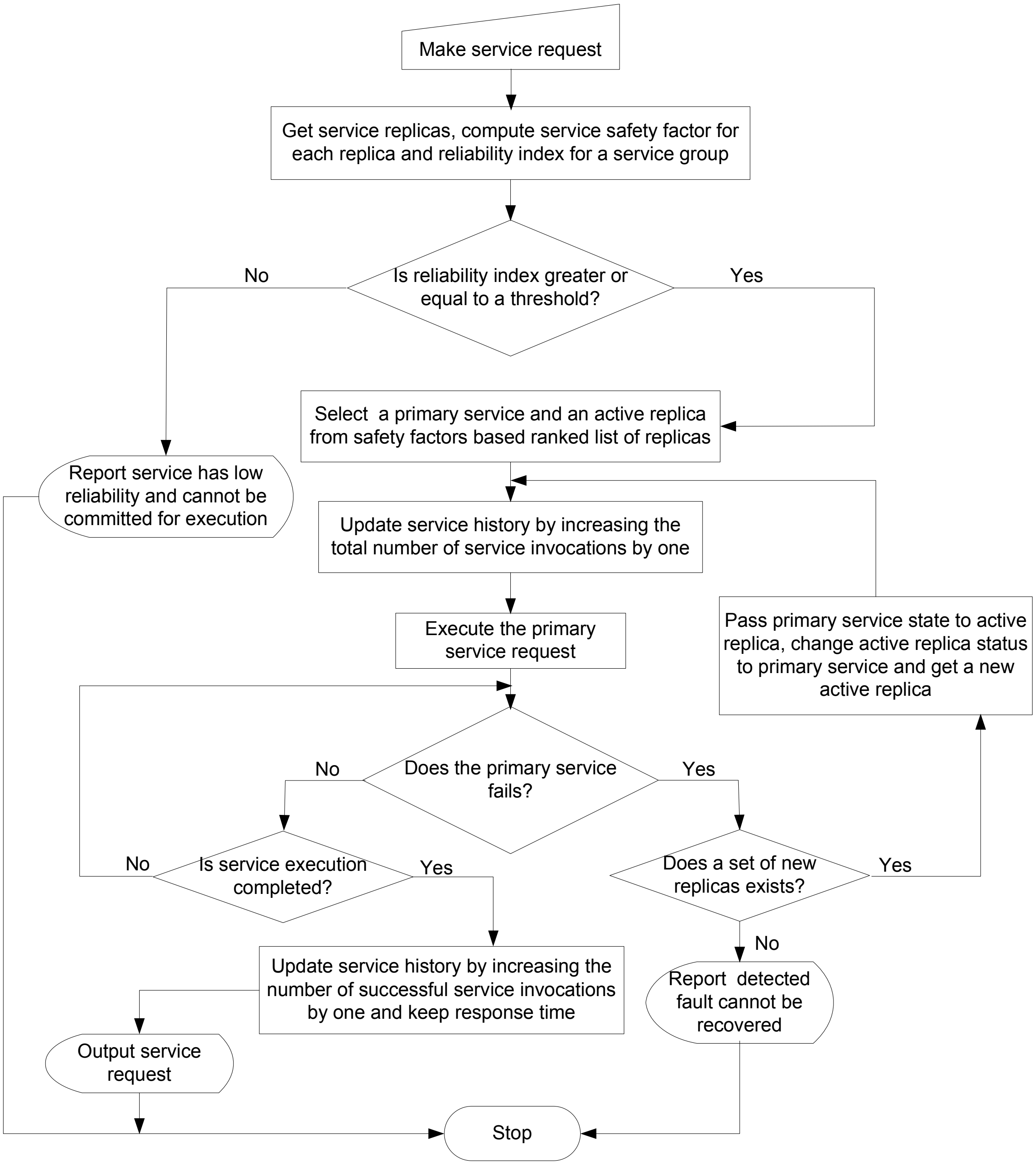

Make service request
Get service replicas, compute service safety factor for each replica and reliability index for a service group
Is reliability index greater or equal to a threshold?
No
Yes
Select a primary service and an active replica from safety factors based ranked list of replicas
Report service has low reliability and cannot be committed for execution
Update service history by increasing the total number of service invocations by one
Pass primary service state to active replica, change active replica status to primary service and get a new active replica
Execute the primary service request
Does the primary service fails?
No
Yes
Is service execution completed?
No
Yes
Does a set of new replicas exists?
Yes
No
Update service history by increasing the number of successful service invocations by one and keep response time
Report detected fault cannot be recovered
Output service request
Stop

Figure 4. 5 Service recovery algorithm

The DSR algorithm commits a service in a service group for an execution only when the group reliability index is greater than a threshold value. A reasonable threshold value is 3, which is above average (=2.5). The higher the reliability index, the more satisfactory is the performance of the services in a group. A reliability index is a probabilistic measure of safety and it represents the number of standard deviations that separates the mean safety factor from the critical safety factor (=1). A safety factor (factor of safety) of a system, usually treated as a random variable is defined as the ratio of capacity to demand for the system. In reality, safety factor is often best fit by a lognormal rather than normal distribution. Thus, the calculation of reliability index (β) is given by:

$$\beta = \frac{\ln\left(\frac{\mu}{\sqrt{1+V^2}}\right)}{\sqrt{\ln\left(1+V^2\right)}} \qquad (4.1)$$

where:

β = lognormal reliability index

μ = mean safety factor

V = coefficient of variation of safety factor (=$\frac{\sigma}{\mu}$)

σ = standard deviation of safety factor

As a result, Service Safety Factor (SSF) is used by our algorithm to rank replicas in a service group. A replica having the highest SSF (=primary service) is assumed to be the most reliable among other replicas in a service group. The primary service is the one first selected for execution before any other replica is selected. If the selected service replica fails, the service with the next higher SSF (=active replica) gets selected for execution and so on. This process of service fault recovery is transparent to the client of the service.

However, when the reliability index of a service group is below a threshold or when all service replicas in a service group fail, the client of the requested service is notified.

The computation of safety factor and reliability index is based on the previous history of service invocations. Let x_1 be number of successful service invocations, x_2 the actual service response time, N the total number of service invocations, $x_3 = aN$ the expected number of successful service invocations, a $(0 < a \leq 1)$ the percentage of the expected number of successful invocations and x_4 the expected service response time. We used multiplicative Cobb-Douglas utility function to represent the Capacity (C) and Demand (D) for a service. The utility function is used in a bounded rational decision-making context, because the fault recovery algorithm recommends a service group with high reliability index and the best service replica (replica with highest safety factor) is selected for execution. Previous research effort (Rand, et. al., 2003) demonstrated that the decision making of agents using Cobb-Douglas utility function can generate distributions of cluster sizes that compare favorably with the structural form of real-world entities. A multiplicative Cobb-Douglas function is also preferred because it eliminates the possibility that a quantity with zero suitability on the factor will have a non-zero utility (Brown and Robinson, 2006). Thus, the capacity of a service is a function of the number of successful service invocations over a period of time. This utility function is defined as:

$$C(x_1, x_2) = x_1^{\alpha} x_2^{1-\alpha} \tag{4.2}$$

Similarly, the demand for a service is a function of the expected number of successful service invocations over a period of time and is defined as:

$$D(x_3, x_4) = x_3^{\beta} x_4^{1-\beta} \tag{4.3}$$

The service safety factor is the utility function U(x) defined as:

$$U(x)=\frac{x_1^{\alpha}x_2^{1-\alpha}}{x_3^{\beta}x_4^{1-\beta}} \tag{4.4}$$

where:

α and β are respectively the output elasticity measures of the capacity and demand for a service and $x=(x_1,x_2,x_3,x_4)$ is the input vector. The selection of α and β can affect the performance of the utility function. Since we expect the capacity to be higher than the demand for a reliable service, α and β should be selected such that $\alpha>\beta$.

4.3.1.2 Service replication manager

The replication manager allows the service group manager to access service replicas either in the local node or remote node. This manager uses WS-Replication to discover service replicas.

4.3.1.3 Service recommender

The recommender is mostly used for databases and user profile preferences, but we also decided to use this recommender to reduce delay during fault recovery. A recommender analyzes the occurrence of all faults in a particular Grid node and recommends services to execute the request or substitute the failed service. This component is an external component that our model depends on.

4.3.1.4 DSR model information storage

The DSR uses the repository for the following activities:

1. Service local information repository is used to store service information.
2. Virtualized registry is a virtualized registry in the network with the help of Grid middleware to discover service replicas.

4.3.2 Model component interaction

During service failure and service provisioning all the components of the DSR model interact. Figure 4.6 shows how the components interact when DSR model receives the request and when the fault occurs during service execution.

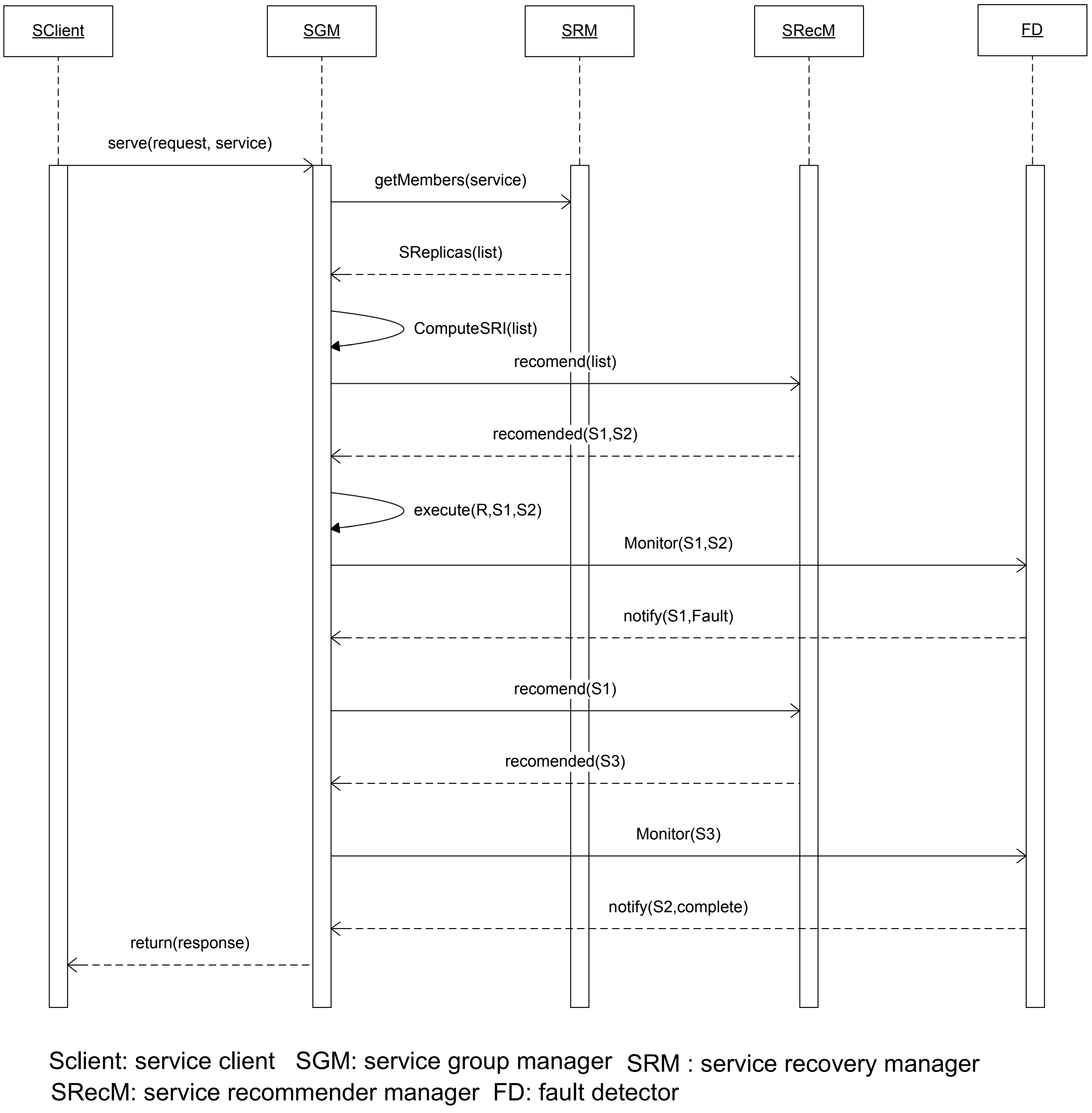


Figure 4. 6 Fault recovery sequence diagram

Our proposed model selects two services with high reliability index to serve the request at a time; one of the services is the primary service while the other one is the active replica. The two services are selected based on their reliability index. Figure 4.7 shows the interaction of the model that is composed of three tiers.

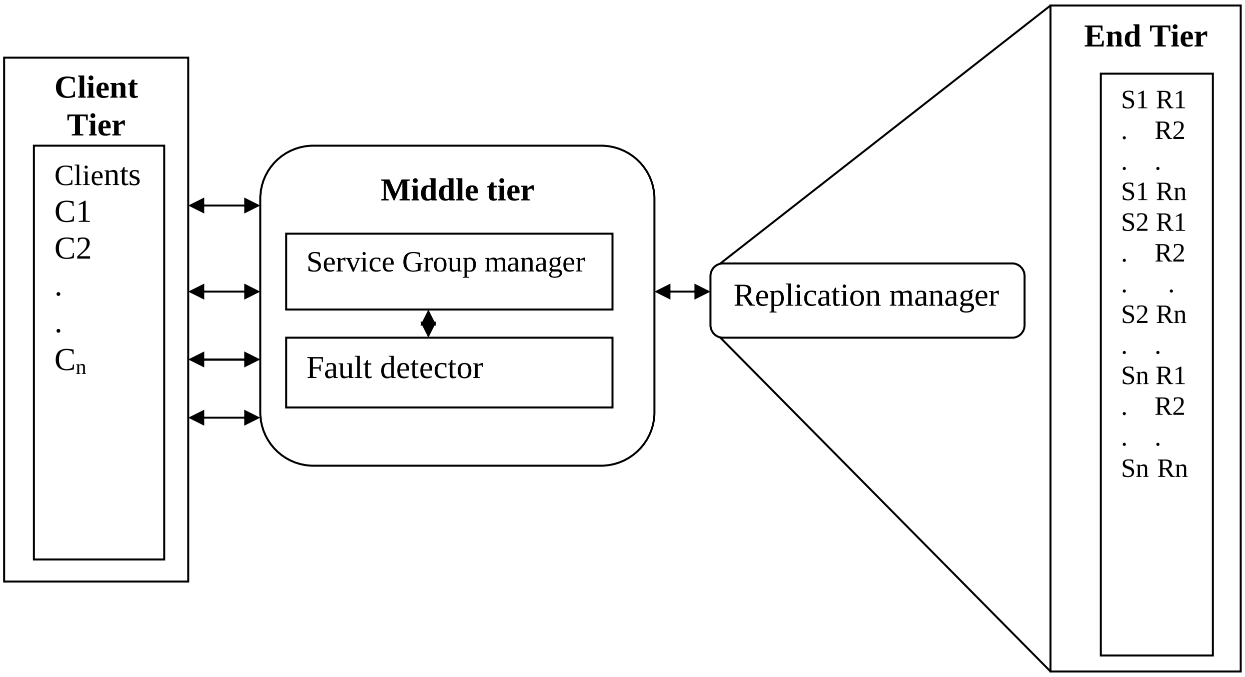
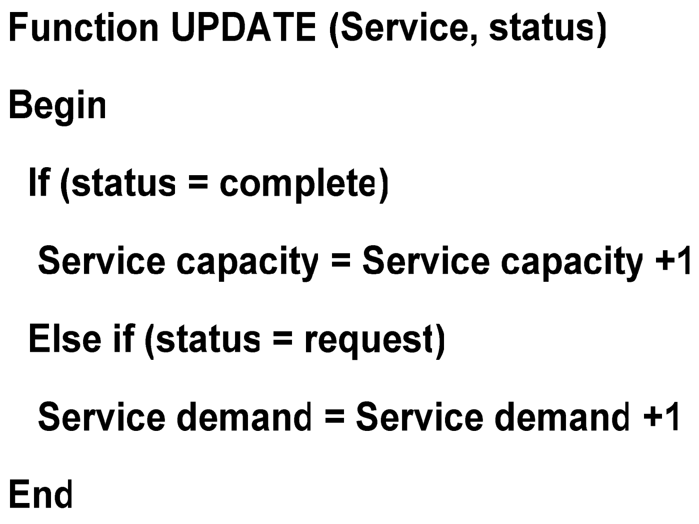


Figure 4. 7 SDR interaction diagram

The Service Group Manager (SGM) receives requests in a FIFO from clients. During the process of serving a request a SGM interacts with the two components namely, Replication manager (RM) and Fault Detector (FD). RM gets the service identifier of the requested service and retrieves information of its replicas. It returns the capacity and demand of each replica to the SGM. Capacity is the number of times a particular service succeeds in processing a request. Demand is the total times of invocations of a service.

The SGM computes the safety factor and reliability index of each service from each service capacity and demand. The SGM selects the replica with the high reliability index, which is also greater than the average reliability index as the primary service and the second highest as an active replica.

During the execution process each running replica gets monitored by the fault detector to check for faults. Whether the replica succeeds or fails, the SGM gets notification to update the replica's history. Figure 4.8 shows the algorithm used to update service history.



```
Function UPDATE (Service, status)
Begin
  If (status = complete)
   Service capacity = Service capacity +1
  Else if (status = request)
   Service demand = Service demand +1
End
```

Figure 4. 8 History updating algorithm

We consider response time as the time from when SGM receives the request to when the SGM passes the response to the service client. The response time in our approach is given by the following equation

$$R_s = C_{nt} + D_{nt} \qquad (4.4)$$

where, C_{nt} is the time it takes to compute reliability index for n replicas and primary and active replica selection, while D_{nt} is the time it takes to receive response. The D_{nt} will vary from the fact that the request can get served by one or more replicas. Also reliability index time computation C_{nt} will vary with the number of replicas.

Performance overhead P_{as} is the time it takes for the service to recover after the web service expected response time has expired. It is given by the following equation

$$P_{as} = (T_n - T_1)/T_1 \quad (4.5)$$

where $T_{(n)}$ is the service response time when the service group has n replicas and $T_{(1)}$ is the response time when no replication is used.

4.4 Summary

The DSR model aims to improve user satisfaction through keeping the service that operates as stated in the SLA. The delay or latency from this research work can be the combination of the following attributes.

i. Mean Time To Recover (MTTR)

ii. The Average Response Time (ART).

The above mentioned attributes would be used to conclude whether DSR model is efficient as far as the delay is concerned. This would help to conclude whether DSR model is reliable or scalable as far as the number of fault is concern and as the number of service replicas increases. This would also show whether DSR model improves service availability and reliability. We assume that immediately the fault has occurred the adopted detector model would detect that fault.

Recalling the goal of this research, service recovery is our main focus, but from the fact that effective service recovery depends on effective fault detection, we then took fault detection into consideration.

From the requirements scenario, The DSR model would be able to automatically recover from all recoverable faults and report accurate failure in a human understandable way for all unrecoverable fault occurrences. It would also log all fault occurrences for the service provider to analyze and have some conclusions. Through automated fault recovery, service downtime would be reduced.

CHAPTER FIVE

MODEL IMPLEMENTATION AND EXPERIMENTATION

5.1 Introduction

This chapter describes the simulation of the proposed autonomic service recovery model presented in the previous chapter. As explained in chapter four, the main goal of the dynamic service recovery model is to develop a dynamic and automated system for service fault management to improve service availability in a Grid environment. The objectives to achieve this were revisited in chapter four in order to ascertain how these were to be achieved in the model design. The model implements the *monitor*, *analyzer*, *planner*, *knowledge*, and *executer* autonomic computing elements. The *monitor* is the mechanism whereby fault messages from the detector component get passed and accepted by the recovery component. A message can be received through the message bus or by message passing between the two components. The *analyzer* then uses fault specification to get fault identification. The *planner* component processes the solution for the identified fault. The *knowledge* component keeps or stores information about fault and recovery plans.

In demonstrating the performance and the behavior of our model, we present some assumptions considered, the description of the simulation, the simulation environment, and the system interface. Finally, we present performance evaluation of the proposed model.

5.2 Basic assumptions of the simulation model

In developing our simulation, the following are assumptions we have considered due to the duration of this project and considerations of the environment where it will function.

a. The grid infrastructure is running, services that are deployed and service client are requesting for services.

b. Each service has a set of replicas deployed in the local and remote nodes.

c. Network behavior is normal and does not fluctuate.

d. Replicas implementation can be different, but gives the same functionality.

5.3 Description of the simulation

The scenario painted in chapter four is considered in simulating our model. Services like Scheduler, Payroll and Billing that operate in fulfilling a particular request were considered. Payroll and Billing services depend on scheduler's information. Due to high complexity and high service demand of such services deployed in a Grid environment, DSR has been proposed to address these challenges.

The Grid environment behavior was also considered in simulating our model. Services dynamically join and leave the network. Services were being provided by different providers with different service characteristics. Services were classified according to their characteristics. The service clients (applications or services) initiate the request to a particular service. Service characteristics for the requested service are then used to query service replicas. Services were being accessed through Service Group Manager (SGM) to ensure service request completion.

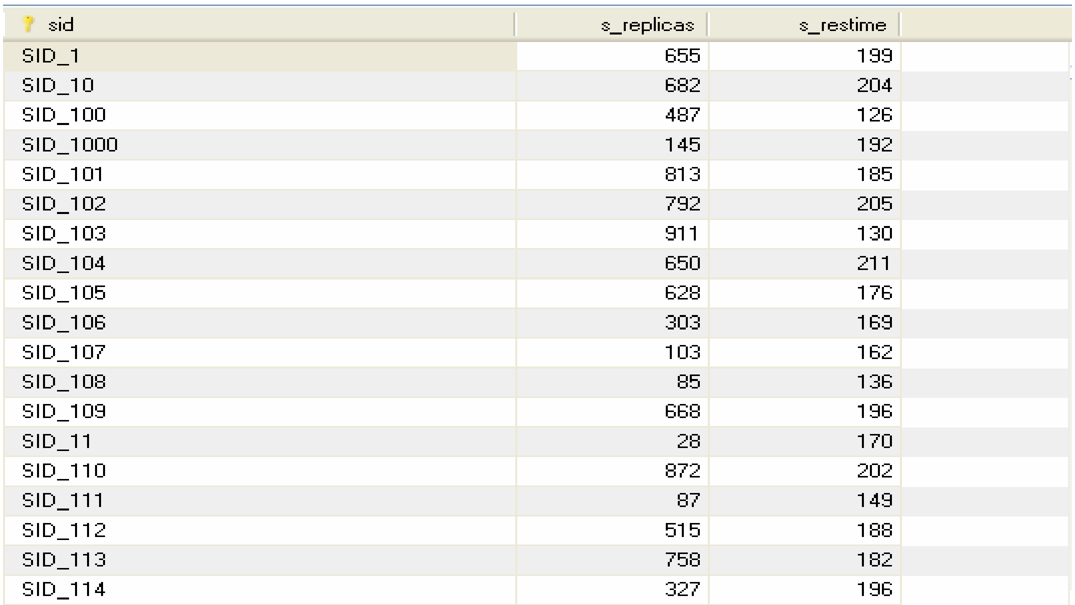
SGM uses a recovery manager to query virtualized service replicas of the requested service. SGM is automatically updated when a service leaves, fails, completes or joins the network. Service policy is used to detect whether the service is behaving normally. Our model monitors service reliability throughout the execution process. This helps in improving service delay when a service becomes unreliable by invoking another service with high reliability to take over. Finally, SGM returns the response to a service client.

The autonomic service recovery model is simulated as a Java application. The object service is simulated with some service properties for example (service name, service response time and service fault) and also with properties that define fault types. From the fact that fault messages get received in a sequence, a service with its properties is randomly generated with faults attached to the service. The faulty service is generated by the thread service generator. After the faulty service is received the object analyzer retrieves the fault from the faulty service. The fault is then passed to the planner object to process and provide the recovery solution. This recovery component gets detected faults from the fault detection model (*Tang et al, 2005).* Figure 3.2 is the overview of the model. Multi-agents are adopted in this model to monitor and manage the occurrence of faults. A policy is used to check whether the monitored service has faults or not.

A replica is a service that performs the same task as the failed service as far as this research is concerned. During the process of replica selection, a reliability index is used.

5.4 Simulation environment

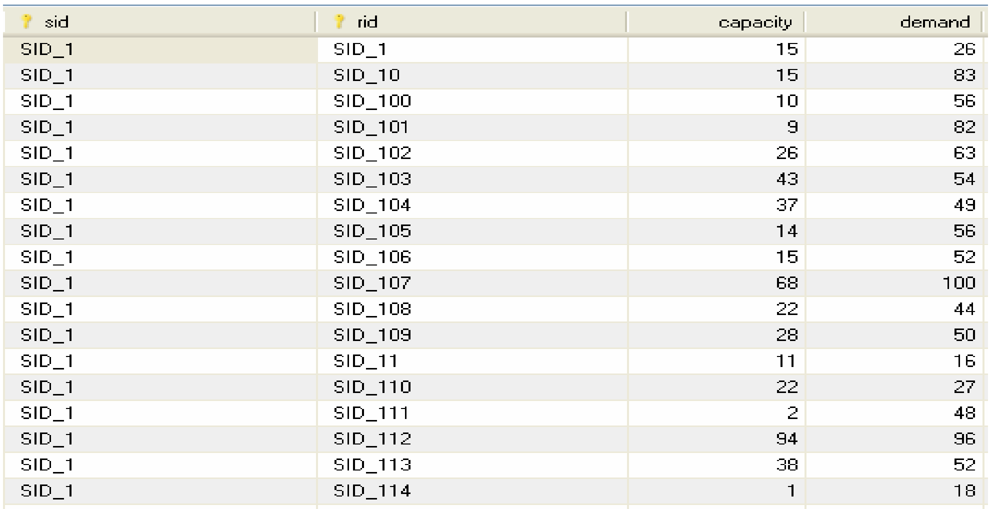
The simulation of our model was carried out in Netbeans 6.1 Integrated Development Environment. Services' faults and plans were stored in the MySQL database. The application was tested on a desktop machine running Windows XP Professional Edition. The machine was an Intel Pentium IV processor with a processing speed of 3 GHz and 512 MB of RAM. The application consumed less than 5.0 MB of hard-disk storage. To show the execution results of our model, we designed an interface that allow the user to interact with the application.



sid	s_replicas	s_restime
SID_1	655	199
SID_10	682	204
SID_100	487	126
SID_1000	145	192
SID_101	813	185
SID_102	792	205
SID_103	911	130
SID_104	650	211
SID_105	628	176
SID_106	303	169
SID_107	103	162
SID_108	85	136
SID_109	668	196
SID_11	28	170
SID_110	872	202
SID_111	87	149
SID_112	515	188
SID_113	758	182
SID_114	327	196

Figure 5. 1 Service table

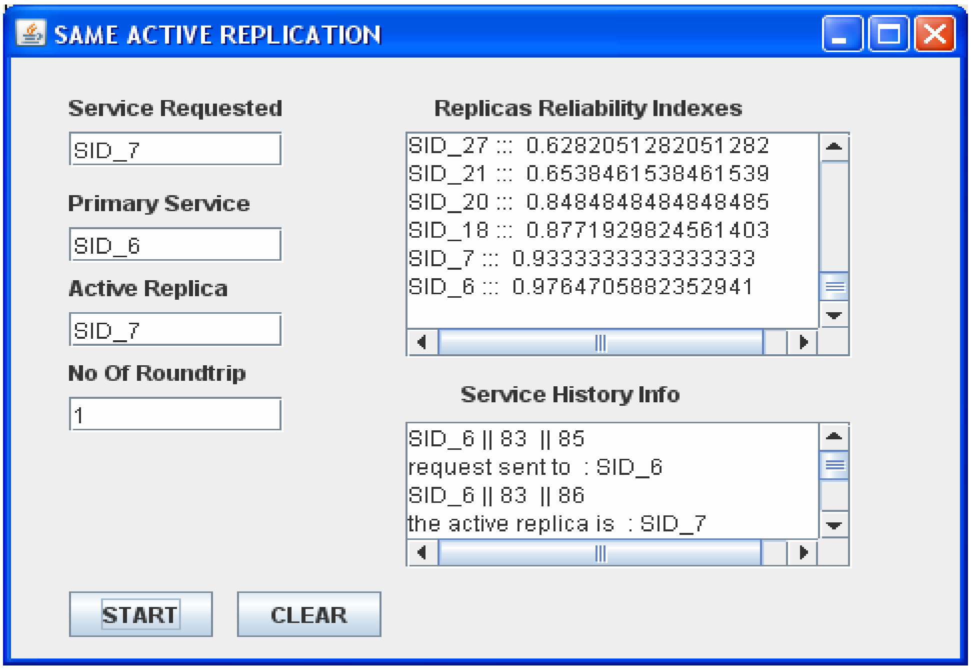
Figure 5.1 shows how services are stored with some service attributes that are more related to this work. In our simulation we generated about 1000 services each with a number of replicas reflected as s_replicas, service identity reflected as sid and the response time in miliseconds reflected as s_restime.



sid	rid	capacity	demand
SID_1	SID_1	15	26
SID_1	SID_10	15	83
SID_1	SID_100	10	56
SID_1	SID_101	9	82
SID_1	SID_102	26	63
SID_1	SID_103	43	54
SID_1	SID_104	37	49
SID_1	SID_105	14	56
SID_1	SID_106	15	52
SID_1	SID_107	68	100
SID_1	SID_108	22	44
SID_1	SID_109	28	50
SID_1	SID_11	11	16
SID_1	SID_110	22	27
SID_1	SID_111	2	48
SID_1	SID_112	94	96
SID_1	SID_113	38	52
SID_1	SID_114	1	18

Figure 5. 2 Replicas table

Figure 5.2 shows a list of service replicas as well as the capacity and demand for each replica. Figure 5.1 and Figure 5.2 show one to many relationship between service and their replicas. This way, we were able to efficiently simulate our model using a simple record structure to represent services and their replicas rather than hierachical structure.



SAME ACTIVE REPLICATION

Service Requested: SID_7

Primary Service: SID_6

Active Replica: SID_7

No Of Roundtrip: 1

Replicas Reliability Indexes

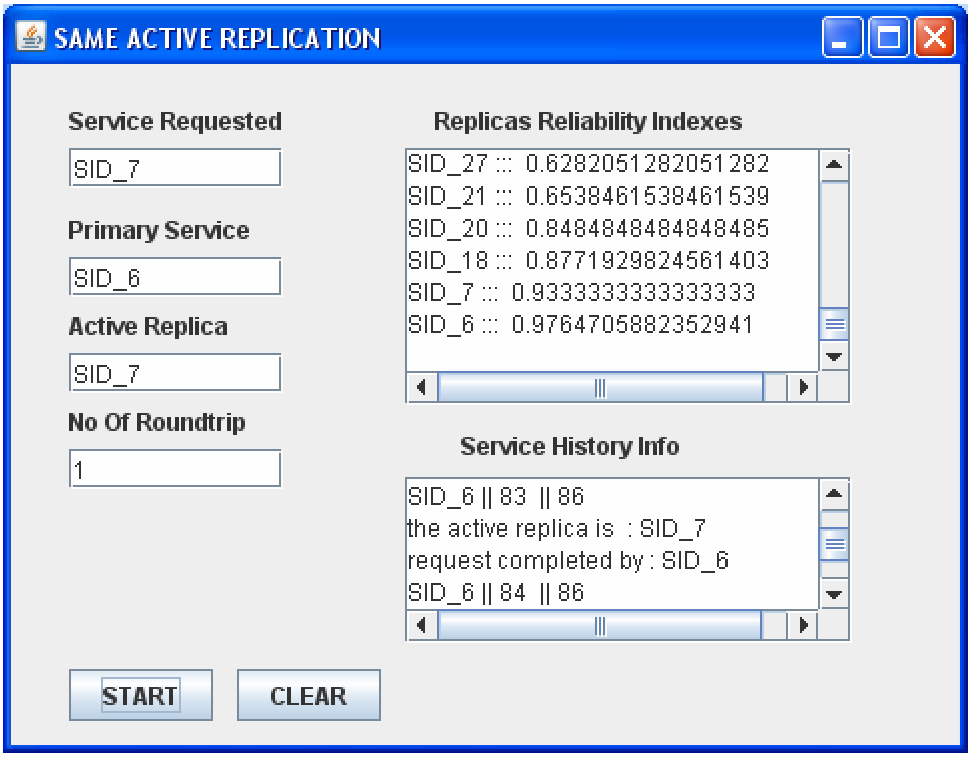
```
SID_27 ::: 0.6282051282051282
SID_21 ::: 0.6538461538461539
SID_20 ::: 0.8484848484848485
SID_18 ::: 0.8771929824561403
SID_7 ::: 0.9333333333333333
SID_6 ::: 0.9764705882352941
```

Service History Info

```
SID_6 || 83  || 85
request sent to  : SID_6
SID_6 || 83  || 86
the active replica is  : SID_7
```

Figure 5. 3 SDR interface A

Figure 5.3 shows a requested service, which is also a member of a replica group. The replica group gets selected and for each replica, a reliability index gets computed. Replicas get sorted in an ascending order according to their reliability indexes. A service with SID_6 is the primary service and service SID_7 is the active replica in this replica group based on high reliability index. Roundtrip indicates the number of replicas that were involved in the execution of a request. Service SID_6 has the demand (number of invocation) of 85 and capacity (number of success) of 83. After SID_6 invocation its demand increases by one.



Service Requested: SID_7

Primary Service: SID_6

Active Replica: SID_7

No Of Roundtrip: 1

Replicas Reliability Indexes

```
SID_27 ::: 0.6282051282051282
SID_21 ::: 0.6538461538461539
SID_20 ::: 0.8484848484848485
SID_18 ::: 0.8771929824561403
SID_7 ::: 0.9333333333333333
SID_6 ::: 0.9764705882352941
```

Service History Info

```
SID_6 || 83  || 86
the active replica is  : SID_7
request completed by : SID_6
SID_6 || 84  || 86
```

Figure 5. 4 SDR interface B

Figure 5.4 shows the service history after a service completes an execution. The service capacity after successful execution has been increased by one from 85 to 86.

5.5 Performance evaluation

In evaluating the performance of our model, we have used response times and performance overhead to compare the performance of active replication model (Liang et al, 2003) against our proposed semi active replication model. In these two models we assume the use of a call back asynchronous interaction pattern during the process of serving a request. The SOAP based protocol WS-Reliability is assumed to be used for exchanging reliable messages among services in both models. The reason for this is to guarantee delivery, to duplicate elimination and message ordering.

Active replication multicasts requests to all replicas with capability to serve the request. In improving response reliability the most common response gets sent to the service client. In eliminating response delays the multicasting component sets the elapsed time after multicasting requests. When that time expires the most common response gets selected from responded replicas, while if all replicas respond before the elapsed time, a common response gets selected. According to Sommerville et al (2006), this model of active replication is called multi-version executing with voting. The response time R_a is given by the following equation.

$$R_a = R_{nt} + E_{nt} + C_{nt} \tag{5.1}$$

where R_{nt} is the time it takes to multicast a request to n replicas, E_{nt} is the time it takes for n replicas to respond or the time set by the multicasting component, C_{nt} is the time it takes to select the most common response from n responded replicas. Performance overhead is given by equation 4.5 in chapter four.

5.6 Experimental results

In conducting our simulation experiments, metrics presented in chapter one (response time and performance overhead) to evaluate our model were used.

The number of faults request and the number of replicas were controlled in the process of evaluating DSR. The performance of DSR is then compared with AR. Parameters of the above mentioned equations and experimental results are also discussed. The elapsed time is assumed to lie between 5 and 25 seconds, service response time is varied between 2 to 20 seconds for each service. When a fault occurs the time it takes for each replica to respond varies from one service to the other. The time was generated randomly around these figures. The number of faults requests varied from 10, 20, 30... 100 requests. The number of replicas also varied from 20, 40, 60... 160 replicas. Capacity and demand were randomly generated between 0 and 1000. It is obvious that the demand will be always less or equal to the capacity because demand depends on capacity. The average response time is given by the sum of n service request response time divided by n and is presented by the following equation for our model

$$AV = \left(\sum_{k=0}^{n} Rs(k)\right)/n \qquad (5.2)$$

While for the active replication is given by the sum of n service request response time divided by n service responses and is presented by the following equation:

$$AV = \left(\sum_{k=0}^{n} Ra(k)\right)/n \qquad (5.3)$$

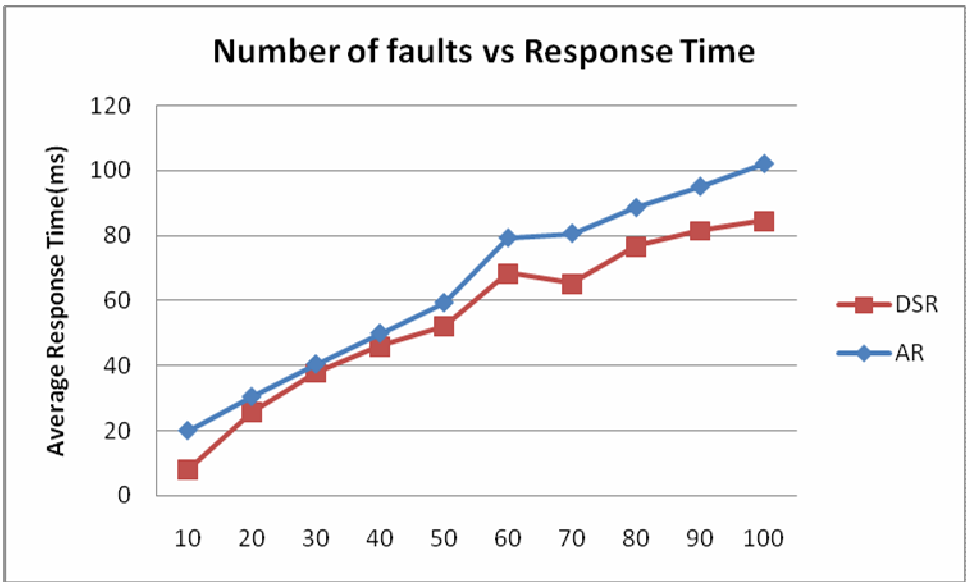


Figure 5. 5 Response time vs number of faults

Figure 5.5 shows the results when 20 replicas were used assuming multicasting and replicas response time is close to zero. Replicas response time is the average time it takes for the number of replicas to respond to a particular request. The fast growth of the DSR graph, in the interval where the number of faults are between 20 and 60, is caused by the number of roundtrips per request during the recovery process. DSR growth went down when the fault requests were 70 because replicas reliability was very high and it then caused a few or no roundtrips. The AR graph grew fastly when the number of fault requests are 60 because of replicas' failures caused the elapsed time to be considered for the number of requests before responding to a request. The AR growth was slower when services are 70 due to the fact that replicas failures were few.

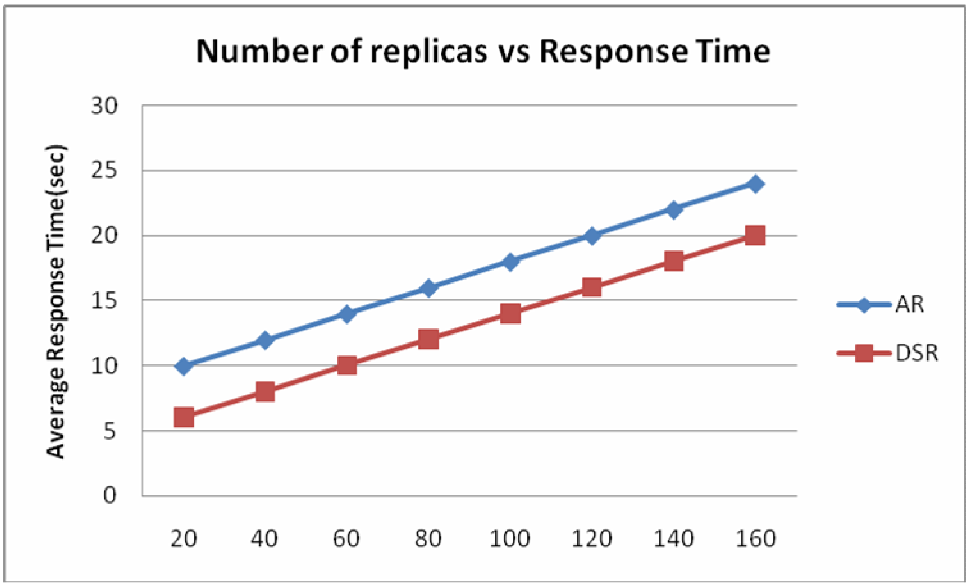


Figure 5. 6 Response time vs number of replicas

The DSR average response time in Figure 5.6 increased because the time to select the primary service and active replica varied along with the number of replicas. Replicas implementation is assumed to be different for a particular service group, this makes the failure of service A not possible in service B. The DSR graph is lower than that of AR because DSR selects two services to serve each request while AR can vary depending on the available replicas. AR response time increased along with the number of replicas because multicasting and replicas response time varies with the number of replicas.

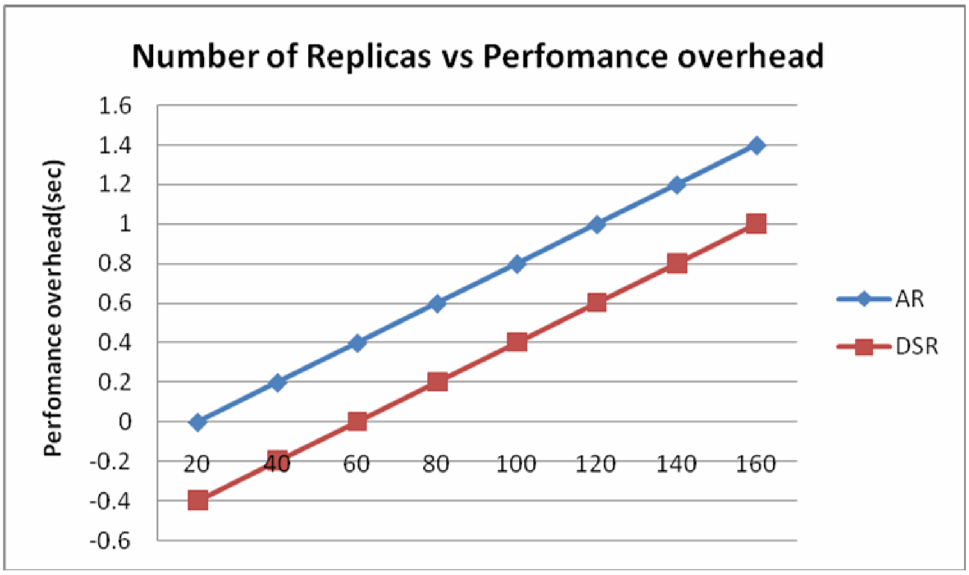


Figure 5. 7 Performance overhead vs number of replicas

Figure 5.7 shows the time it takes for AR and DSR to recover for each set of replicas. The average time to respond given a set of replicas was considered. Looking at the DSR it performed well when the numbers of replicas was less or equal to 60 while AR performed well when the number of replicas was less than 21. The reason was because DSR always have two services that are serving request at a time while the AR can vary. DSR does not have an overhead when services are less or equal to 60, which means the service still returns the response before the response time of the failed service expires. The overhead from both models increased along with the number of replicas. This was owing to the fact that for the DSR, time to select both primary and active replicas increase when the number of services increase. The AR overhead increased because the time to multicast a request increases when the number of services increase and also the average response time increased along with the number of replicas.

5.7 Summary

In summary, from the detailed experiments conducted, DSR performs better than the AR when some replicas have high reliabilities in a service group, while AR performs well when the all replicas have high reliabilities and without service failures during service execution. Our recovery approach efficiently consumes resources than AR approach due to the fact that when a service has 100 replicas in our approach, only two reliable services will serve an incoming request, while in AR all the 100 replicas will serve the request.

CHAPTER SIX

CONCLUSION AND FUTURE WORK

6.1 Introduction

The goal of this research was to develop a dynamic service recovery (DSR) model for Grid services that integrates into GUISET Grid middleware. We have developed the DSR model that effectively improves service availability during service failure. The DSR model makes use of a policy based fault detector model proposed by Tang (2005) to monitor and detect service faults. In order to improve transparency during service recovery, the autonomic computing self healing approach was incorporated in our model. The simulation of the DSR model was used to evaluate our service recovery approach.

This chapter reviews the DSR model developed in this research. In section 6.2 we highlight the achievements of this research. A reasonable critique of the work is also given. In section 6.3, we discuss the limitations of this work and give some suggestions on how the model can be extended in future.

6.2 Summary

There is a very high demand of service availability for Grid enterprise applications. A dynamic service recovery (DSR) model has been proposed to improve service availability and to reduce service failure during service execution. The DSR model offers effective runtime service recovery and is bound to improve service availability through the use of replication approach. In improving the performance of our replication approach and service trustworthiness, a reliability index is used to dynamically select two services with the highest reliability to serve an incoming request. In the two selected services, a service with higher reliability than the other acts as a primary service whiles the other one act as an active replica. With the reliability index we managed:

- To improve resource usage during service execution and reduce delay,
- To reduce service failure during service execution, through the use of service reliability index.

Furthermore, Dai et al (2007) presented that there are a number of faults that constitute service failure like the blocking failure, time-out failure, matchmaking failure, network failure, and program and resource failure. In our model, we managed to reduce failures such as blocking failure, time-out failure, and program and resource failure. The DSR reduced time out failure due to the fact that two most reliable services get selected to serve a request which makes it transparent to the service client when a service fails. To also address time out failures, the reliability index serves as indicator to determine whether a reliable service exists to serve that particular request before committing to a service request. The blocking failure gets reduced from the fact that a service has one or more replicas that can assist if the service fails to serve the request. The completion of execution is guaranteed if there are a number of replicas with reliability index greater than average. The program and resource failure is reduced because replica implementation can be different from one replica to the other. Another fact that reduced this kind of failure is that service replicas could be distributed in the local node and remote nodes.

A comparative analysis of the proposed approach against the active replication approach has been presented.

The results have revealed that the new proposed approach exhibits superior performance characteristics especially when there are services with high and low reliability.

6.3 Limitations and Future Work

The results obtained from the simulation confirmed the suitability of the proposed model for Grid enterprise services. However, the simulation is only an approximation of the reality; therefore, another primary goal in the future work is to observe the behavior of the proposed approach in real life environment. A test-bed Grid network with replicated services could be constructed, with the proposed recovery approach.

This research concentrated on service recovery during service execution; however our approach also pointed to some relevant issues that need to be addressed in the process of service recovery. First is the issue of service provider satisfaction in the process of interchanging service when it comes to service cost. This would need development of a mechanism for distributing quotas among different providers with different pricing schemes. This mechanism would need to be taken into consideration during replication selection. Second is the issue of security among replicas from different providers. We would also like to consider other QoS metrics to evaluate the performance of our approach and to incorporate other recovery mechanisms, for example, mechanisms for recovering from Byzantine faults, matchmaking failures and network failures.

BIBLIOGRAPHY

Adigun, M., Emuoyibofarhe, O., and Migiro, S. (2006). *Challenges to Access and Opportunity to use SMME enabling Technologies in Africa*, a presentation at the 1st All Africa Technology Diffusion Conference, Johannesburg South Africa.

Affaan, M., and Ansari, M. (2006). *Distributed Fault Management for Computational Grids*, Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC'06), pp: 363 - 368 , Washington, DC, USA.

Ahmed, S., Sharmin, M., and Ahamed, S. (2007). *ETS (Efficient, Transparent, and Secured) Self-healing Service for Pervasive Computing Applications*, International Journal of Network Security, Vol.4, No.3, pp: 271–281.

Ardissono, L., Furnari, R., Goy, A., Petrone, G., and Segnan,M. (2006). *Fault Tolerant Web Service Orchestration by Means of Diagnosis*, In Proceedings of the European Workshop. on Software Architectures EWSA'2006), pp: 2-16.

Arshad, N., Heimbigner, D., and Wolf A. L. (2004). *A Planning Based Approach to Failure Recovery in Distributed Systems*, In Proceedings of the ACM SIGSOFT International Workshop on Self-Managed Systems (WOSS'04), pp: 8 -12, ACM Press .

Baresi, L.,Ghezzi, C., and Guinea, S.,(2004), *Smart monitors for composed services*, In ICSOC '04, In Proceedings of the 2nd international conference on Service oriented computing, pp: 193–202, New York, NY, USA, ACM Press

Baresi, L., Guinea, S., Pasquale, L. (2008). *Towards a unified framework for the monitoring and recovery of BPEL processes*, In Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications, pp: 15-19, Seattle, Washington.

Benharref, A., Glitho, R., and Dssouli, R. (2005). *A web service based-architecture for detecting faults in web services*, IM 2005 - *IFIP/IEEE International Symposium* on Integrated Network Management, Vol: 1, pp: 1273-1276.

Brown, D., and Robinson D. (2006). Effects of heterogeneity in residential preferences on an agent-based model of urban sprawl. *Ecology and Society* 11(1): 46. Conference on Distributed Computing Systems (ICDCS'2001), Phoenix, Arizona, USA, April 2001. IEEE Computer Society

Bruning, S., Weißleder, S., and Malek, M. (2007). *A Fault Taxonomy for Service-Oriented Architecture*, 10th IEEE High Assurance Systems Engineering Symposium, pp: 367-368.

Chan, K., Bishop, J., Steyn, J., Baresi, L., and Guinea, S. (2007). *A Fault Taxonomy for Web Service Composition*, in Proceedings of the 3 rd International Workshop on Engineering Service Oriented Applications (WESOA'07), Springer LNCS.

Cook, B., Babu, S., Candea, G., and Duan, S. (2007). *Toward Self-Healing Multitier Services*, International Workshop on Self-Managing Database Systems (SMDB), in conjunction with ICDE-2007, Istanbul, Turkey.

Dabrowski, C., Mills, K., and Rukhin, A.(2003). *Performance of Service-Discovery Architectures in Response to Node Failures*, Software Engineering Research and Practice , pp: 95-104.

Dai, Y., Wang, X., (2006), O*ptimal resource allocation on grid systems for maximizing service reliability using a genetic algorithm*, Reliability Engineering and System Safety, Vol:91 , pp:1071–1082

Dai, Y., Wang , X., and Xie, M.(2005). A virtual modeling and a fast algorithm for Grid service reliability, *The 11th IEEE Pacific Rim Symposium on Dependable Computing* (PRDC2005), pp: 219-227, IEEE Computer Press, China.

DEPARTMENT OF THE ARMY,(1997). *Introduction to probability and reliability methods for use in geotechnical engineering*, Retrieved on September, 19,2008 http://www.usace. army.mil/publications/eng-tech-ltrs/etl1110-2-547/entire.pdf.

Duan, R., Prodan, R.,and Fahringer, T. (2006). Data Mining-based Fault Prediction and Detection on the Grid, In *International Symposium on High Performance Distributed Computing, IEEE Computer Society Press*, pp: 305-308 , Paris, France.

Fang , C., Liang , D., Lin, F., and Lin, C. (2007) .*Fault tolerant Web Services*, Journal of Systems Architecture: the *EUROMICRO Journal*, v.53 n.1, pp: 21-38.

Foster, I. (2006). *Globus toolkit version 4: Software for service-oriented systems*, *Journal of Computational Science and Technology*, Vol: 21, pp:523–530.

Foster, C., Kesselman, J., Nick, M., and Tuecke, S. (2002). *Grid Services for Distributed System Integration*, *IEEE Computer*, vol: 35, pp: 37-46, IEEE Computer Society Press, Los Alamitos, CA, USA.

Fugini, M., and Mussi, E. (2006). *Recovery of Faulty Web Applications through Service Discovery*, SMR-VLDB Workshop, Matchmaking and Approximate Semantic-based Retrieval: Issues and Perspectives, 32nd International Conference on Very Large Databases, Seoul, Korea, pp: 67-80.

Gadgil, H., Fox, G., Pallickara, S., and Pierce, M. (2007). *Scalable, Fault-tolerant management in a Service Oriented Architecture*, In Proceedings of HPDC'07 , pp: 235-236 ,Austin, TX, USA.

Ghosh , D., Sharman , R., Rao, H., and Upadhyaya , S. (2007). *Self-healing systems – survey and synthesis*, *Decision Support Systems*, Vol: 42, pp:2164–2185, Elsevier Science Publishers B. V. Amsterdam, The Netherlands, The Netherlands .

Gokhale, S., and Dasarathy, B. (2007). *Performance Analysis of CORBA Replication Models* ,In Proceedings of the *Third IEEE International Symposium* on Dependable, Autonomic and Secure Computing (DASC 2007) , Vol: 00,pp: 100 – 107, IEEE Computer Society.

Grishikashvili, E.,Pereira, R., and Taleb-Bendiab, A. (2006). Performance evaluation for self-healing distributed services and fault detection mechanisms, *Journal of Computer and System Sciences*, Vol: 72 pp: 1172-1182.

Guinea, S., and Ghezzi, C. (2005). *Self-healing Web service compositions*, International Conference on Software Engineering Proceedings of the 27th international conference on Software engineering (ICSE 2005), pp: 655-655.

Hanemann, A., Sailer, M., and Schmitz, D. (2004). *Assured service quality by improved fault management - service-oriented event correlation*. In Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC04), New York City, New York, USA, ACM.

Hariri, S., Khargharia, B., Chen, H., Yang, J., and Zhang, Y. (2006). The Autonomic Computing Paradigm, *Cluster Computing* , Vol: 9, pp: 5–17, United States.

Hasselmeyer, P. (2005).*On Service Discovery Process Types* 3rd International Conference On Service Oriented Computing (ICSOC '05) , pp: 144-157.

Huda, M., Schmidt,H., and Peake,I. (2005). *An Agent Oriented Proactive Fault-tolerant Framework for Grid Computing*, In Proceedings of the First International Conference on e-Science and Grid Computing(E-SCIENCE), pp: 304-311, Washington, DC, USA.

Huhns, M., and Singh, M . (2005). Service-Oriented Computing:Key Concepts and Principles, *IEEE Internet Computing*, Vol: 9, pp: 75 – 81.

IBM.(2004). *An architectural blueprint for autonomic computing*. Retrieved September, 12, 2007 ,Home-Page: http://www-3.ibm.com/ autonomic /pdfs/ACwpFinal.pdf .

Jiang, M., Zhang, J., Raymer, D., and Strassner, J. (2007). *A Modeling Framework for Self-Healing Software Systems*, Retrieved on May ,12,2008, http://www.comp.lancs.ac.uk/~bencomo/MRT07/papers/MRT07_Jiangl_etall.pdf.

Kim , B., and Hariri, S. (2007). *Anomaly-based Fault Detection System in Distributed System*, In Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA), pp: 782-789, Washington, DC, USA.

Kephart, J., and Chess, D.(2003). The Vision of Autonomic Computing, *IEEE Computer* 36(1), pp: 41-50.

Lee, H., Chin, S., Lee, J., Lee, D., Chung, K., Jung, S., and Yu, H. (2004)." *A Resource Manager for Optimal Resource Selection*",IEEE International Symposium on Cluster Computing and the Grid.

Lee, Y., Oh, J., and Han. S. (2005). Enriching Quality and Fault-Tolerance of Web Services System, *International Journal of Web Services Practices*, Vol.1, No.1-2 pp: 153-157.

Li, M., and Baker, M. (2005). *The Grid Core Technologies* , place: John Wiley & Sons Ltd, ISBN-10 0-470-09417-6 (PB).

Liang, D., Fang, C., Chen, C.,and Lin, F. (2003).*Fault tolerant web service*,In Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference (APSEC'03),pp:310-323, Washington, DC, USA.

Liu, H., Bhat, V., Parashar, M., and Klasky,S.(2005). *An Autonomic Service Architecture for Self-Managing Grid Applications*, Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005), pp: 132 - 139, Seattle, WA, USA, IEEE Computer Society Press.

Maheshwari, P., and Tam, S. (2006).*Events-Based Exception Handling in Supply Chain Management using Web Services*, Digital Object Identifier 10.1109/AICT-ICIW.2006.93, pp: 151- 151.

Martinello, M., Kaaniche, M., and Kanoun, K. (2005). Web Service Availability—Impact of Error Recovery and Traffic Model, In *Reliability Engineering and System Safety*, Elsevier, 89(1), pp:6-16.

Maximilien, E.,and Singh, M. (2003). *Agent-based architecture for autonomic web service selection*, In Workshop on Web Services and Agent-based Engineering at Autonomous Agents and Multi-Agent Systems (WSABE'2003).

Menasc´e, D., and Casalicchio, E. (2004). *Quality os Service Aspect and Metrics in Grid Computing*, Int. CMG Conference 2004, pp: 521-532.

Mikic-Rakic, M., Mehta, N., and Medvidovic, N. (2002). *Architectural style requirements for self-healing systems*, In Proceedings of the first workshop on Self-healing systems(WOSS '02), pp: 49-54,Charleston, South Carolina, USA.

Modafferi, S., Mussi, E., and Pernici, B.(2006). *SH-BPEL: a self-healing plug-in for Ws-BPEL engines*, In *ACM Proceedings of the 1st workshop on Middleware for Service OrientedComputing (MW4SOC'2006)*, pp: 48-53, Melbourne, Australia.

Naccache, H. and Gannod, G.(2007). A Self-Healing Framework for Web Services, In *Proceedings of the 2007 IEEE International Conference on Web Services*, pp: 1-8.

Neti, S.; and Muller, H. (2007). *Quality Criteria and an Analysis Framework for Self-Healing Systems*, Software Engineering for Adaptive and Self-Managing Systems ICSE Workshops SEAMS apos;07, *IEEE Computer Society* Washington, DC, USA .

Parashar, M., and Hariri, S. (2005). *Autonomic computing: An overview*, In J.-P. B. et al., editor, *Unconventional Programming Paradigms*, vol: 3566, pp: 247–259, Mont Saint-Michel: Springer Verlag.

Parashar, M., and Hariri,S. (2007). *Autonomic Computing Concepts, Infrastructure,and Applications,* place: Taylor & Francis Group, LLC, ISBN-10: 0-8493-9367-1 (Hardcover).

Pereira, E., Pereira, R., and Taleb-Bendiab, A. (2006). Performance evaluation for self-healing distributed services and fault detection mechanisms, *Journal of Computer and Systems Science*, ELSEVIER, pp:492-502.

Qian-mu, L., Man-wu, X., and Hong Z.(2006). *A Root-fault Detection System of Grid Based on Immunology*, In Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC), pp: 369 - 373, Washington, DC, USA.

Rand, W., Brown D., Page S., Riolo R., Fernandez L., and Zellner M. (2003). Statistical validation of spatial patterns in agent-based models. *In* J. P. Muller, editor. *Proceedings of Agent Based Simulation 4* (Montpellier, 2003). CIRAD, Montpellier, France.

Rott, A. (2007). *Self-Healing in Distributed Network Environments*, In Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW), Vol: 01, pp: 73-78, Washington, DC, USA.

Shin, M., and Hoon An, J. (2006). *Self-Reconfiguration in Self-Healing Systems*, In Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06),pp: 89 - 98 , Washington, DC, USA.

Tanenbaum, A., and Van Steen, M.(2007). *Distributed Systems principles and paradigms* (2 ed), place: Pearson Prentice Hall, ISBN : 0-13-239227-5.

Tang, J. (2006). *Supporting Fault Tolerance in Dynamic Management of Workflow-Oriented Services*, In Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW), pp: 152- 158, IEEE Computer Society Washington, DC, USA .

Tang, J., Zhou, B., He, Z., (2005), *Policy driven and multi-agent based fault tolerance for Web services*, *J Zhejiang Univ SCI 2005 6A(7), pp:676-682*

Tartanoglu, F., Issarny, V., Romanovsky, A.,and Levy,N. (2003). *Coordinated Forward Error Recovery for Composite Web Services*, In Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03) , pp: 167-176.

Tian, W., Zulkernine, F., Zebedee, J., Powley, W.and Martin, P. (2005). *An Architecture for an Autonomic Web Services Environment*, In Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems WSMDEIS (ICEIS 2005), pp: 54-66, Miami, Fl.

Townend, P., Xu, J.(2003). Fault *Tolerance within Grid environment*, In Proceedings of the UK e-Science All Hands Meeting (AHM2003), pp: 272-275.

Treaster, M. (2005). A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems, *ACM Computing Research Repository* (CoRR), Vol: 501002, pp: 1-11.

Yoshikawa, T., Ohta, K., Nakagawa, T., and Kurakake, S.(2003). *Mobile Web Service Platform for Robust, Responsive Distributed Application*, In Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03), pp: 144 - 148, Washington, DC, USA .

Zeid, A., and Gurguis, S. (2005). *Towards autonomic web services*, Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), pp: 1-5.

APPENDIX

A.1 Service Recovery Algorithm Implementation

```
package faultrecovery;

import java.util.ArrayList;


/**
 *
 * @author Sihle
 */
public class SReplication {
Service service = new Service();
Faults fault = new Faults();
SortObjects so = new SortObjects();
ServiceDB db = new ServiceDB();
Time t = new Time();
int pprimary ;
int preplica ;
long latency = 0;
ArrayList rel = new ArrayList();
boolean completed = false;
public ArrayList SActiveReplication()
{


 ArrayList messages = new ArrayList();


 String gsid = this.generateRequest();
 ArrayList repo  = db.retrieveReplicas(gsid);
 totalproduct = repo.size();
 pprimary = repo.size()-1;
 preplica = repo.size()-2;
 rel = this.computeReliability(repo);
 String psid = this.getRID(rel, pprimary);
 String ssid = this.getRID(rel, preplica);
```

```
messages.add("requested service : "+ gsid);
messages.add("request sent to  : "+ psid);
messages.add("the active replica is  : "+ ssid);
double normal = Normal(repo);
double lnormal = LogNormal(repo);
messages.add(normal+"  normal : lnormal   "+lnormal);
this.updateDemand(repo, psid);
while( completed== false)
{
   if(preplica > 0)
    {
      double test = Math.random();

        if(test > 0.7)
           {
            messages.add("request completed by : "+ psid);
            updateSuccess(repo, psid);
                        latency = t.getTime() - ((Replicas)repo.get(0)).getStart_Time();

            completed = true;
           }
        else
          {
            double failure = Math.random();
            if(failure  < 0.15)
            {
             preplica = preplica -1;
             pprimary = pprimary -1;
                        messages.add("the new primary service is taking over : "+
                        ssid);
             psid = ssid;
             ssid = getRID(rel, preplica);
             messages.add("new active replica is  : "+ ssid);
            }
          }
   }
   else
   {
      messages.add("error : failure needs human intervention");
      latency = t.getTime() - ((Replicas)repo.get(0)).getStart_Time();
      completed = true;
```

```
     }
  }
 return messages;
}


public ArrayList computeReliability(ArrayList reposit)
{


 for(int j=0 ; j < reposit.size();j++)
 {
  Replicas r = (Replicas)reposit.get(j);
  double reliable = (double)(((double)r.getCapacity()/(double)r.getDemand()));
  System.out.println(r.getRid()+"   "+ reliable);

  r.setReliability(reliable);
  Object ob = reposit.set(j, r);


 }
 so.sort(reposit);

 return reposit;
}

public String getRID(ArrayList repos, int position)
{

    Replicas r = (Replicas)repos.get(position);

 return r.getRid();
}

public void updateDemand(ArrayList repo, String rid)
{
  for(int u = 0; u < repo.size(); u++)
  {
   Replicas r = (Replicas)repo.get(u);
   if(r.getSid().equals(rid))
   {
    int value = r.getCapacity() + 1;
    r.setCapacity(value);
    repo.set(u, r);
    u = repo.size();
   }
  }
 }

public void updateSuccess(ArrayList repo, String rid)
{
  for(int u = 0; u < repo.size(); u++)
  {
   Replicas r = (Replicas)repo.get(u);
   if(r.getSid().equals(rid))
   {
    int value = r.getCapacity() + 1;
    r.setCapacity(value);
    repo.set(u, r);
    u = repo.size();
   }
  }
 }

    public int getTotalproduct() {
        return totalproduct;
    }

    public void setTotalproduct(int totalproduct) {
        this.totalproduct = totalproduct;
    }
    public int getPprimary()
    {
     return pprimary;
    }
     public int getPreplica()
    {
     return preplica;
    }
   public void populateData(int noOfServices)
   {
     db.initialize();
     ArrayList services = service.generateServices(noOfServices);
     for(int i =0; i< services.size();i++)
     {
      Service s =(Service)services.get(i);
       if(i < s.getNumberOfReplicas())
       {
       int demand1 = 1+(int)(Math.random()*100);
       int capacity1 = ((int)(Math.random()*100))%demand1;
       db.insertReplica(s.getService_ID(), s.getService_ID(), capacity1, demand1,0);
       }

       db.insertService(s.getService_ID(), s.getNumberOfReplicas(),
(int)s.getRes_time());
       ArrayList replica = service.generateReplicas(s.getNumberOfReplicas());

       for(int j=0;j < replica.size();j++)
       {
           Service s1 =(Service)replica.get(j);
           int demand = 1+(int)(Math.random()*100);
           int capacity = ((int)(Math.random()*100))%demand;
           db.insertReplica(s.getService_ID(), s1.getService_ID(), capacity, demand,
0);
       }
     }
   }
   public String generateRequest()
   {
     int no = 1 + (int)(Math.random()*1000);
     return "SID_"+no;
   }

   public ArrayList getCD(ArrayList list)
{
 ArrayList cd = new ArrayList();
 for(int i = 0; i< list.size(); i++)
{
 Replicas r = (Replicas)list.get(i);
 int b = r.getDemand()/ r.getCapacity();
 cd.add(b);
}
return cd;
```

```
}

public double getmean(ArrayList list)
{
 int mean = 0;
  for(int i =0; i < list.size(); i++)
{
 mean = mean +Integer.parseInt(list.get(i).toString());
}
 double rmean = (double)mean/(double)list.size();
 return rmean;

}


public double getmeanC(ArrayList list)
{
 int mean = 0;
  for(int i =0; i < list.size(); i++)
{
 Replicas r = (Replicas)list.get(i);
 mean = mean + r.getCapacity();
}
 double rmean = (double)mean/(double)list.size();
 return rmean;

}

public double getmeanD(ArrayList list)
{
 int mean = 0;
  for(int i =0; i < list.size(); i++)
{
 Replicas r = (Replicas)(list.get(i));
 mean = mean + r.getDemand();
}
double rmean = (double)mean/(double)list.size();
 return rmean;

}
```

```
public double getQc(double mean, ArrayList list)
{
 double c = 0;
 for(int i =0; i < list.size(); i++)
{
Replicas r = (Replicas)(list.get(i));
 c = c + Math.pow((double)(r.getCapacity() - mean),2);
}
double x = (double) c/(double)(list.size()-1);
return Math.sqrt(x);
}
public double getQd(double mean, ArrayList list)
{
double c = 0;
 for(int i =0; i < list.size(); i++)
{
 Replicas r = (Replicas)(list.get(i));
 c = c + Math.pow(((double)r.getDemand() - mean),2);
}
double x = (double)c/(double)(list.size()-1);
return Math.sqrt(x);
}
public double Normal(ArrayList list)
{
 double b =
getmean(getCD(list))/Math.sqrt((Math.pow(getQc(getmeanC(list),list),2) +
(Math.pow(getQd(getmeanD(list),list),2))));
 return b;
}
public double LogNormal(ArrayList list)
{
 double a = getmeanC(list)*
Math.sqrt(1+Math.pow((getQd(getmeanD(list),list)/getmeanD(list)),2));
 double b = getmeanD(list)*
Math.sqrt(1+Math.pow((getQc(getmeanC(list),list)/getmeanC(list)),2));
 double c = Math.log(b/a);

return c;
}
}
```

A.2 Service Implementation

```
package faultrecovery;

import java.util.ArrayList;

/**
 *
 * @author Sihle
 */
public class Service {

    /** Creates a new instance of Service */
    private String service_ID;
    private int clients;
    private long res_time;
    private Faults fault;
    private int numberOfReplicas;
    private int numberOfLReplicas;
    ArrayList faults = new ArrayList();
    Faults faulty = new Faults();
    private double reliability;
    private long start_time;
    Time time = new Time();
    public Service() {
    }

    public String getService_ID() {
        return service_ID;
    }

    public void setService_ID(String service_ID) {
        this.service_ID = service_ID;
    }

    public long getRes_time() {
        return res_time;
    }
    public void setRes_time(long res_time) {
        this.res_time = res_time;
    }

    public Faults getFault() {
        return fault;
    }

    public void setFault(Faults fault) {
        this.fault = fault;
    }

     public ArrayList generateServices(int numberOfService)
    {
     ArrayList gfaults = new ArrayList();
     for(int i = 1; i <= numberOfService; i++)
     {
        Service s = new Service();
        s.setStart_time(time.getTime());
        int nor = 1 + (((int)(Math.random()*1000)));
        long rtime = 120 + (((int)(Math.random()*100)));
        int nolr = 1 + (((int)(Math.random()*10))% 3);
        int nclients = 1 + (((int)(Math.random()*50)));
        String sid = "SID_"+i;
        s.setService_ID(sid);
        fault = s.generateFault();
        s.setFault(fault);
        s.setNumberOfReplicas(nor);
        s.setNumberOfLReplicas(nolr);
        s.setRes_time(rtime);
        s.setClients(nclients);
        gfaults.add(s);
     }
     return gfaults;
    }


public ArrayList generateReplicas(int numberOfService)
    {
     ArrayList gfaults = new ArrayList();
     for(int i = 1; i <= numberOfService; i++)
     {
        Service s = new Service();
        s.setStart_time(time.getTime());

        int nor = 1 + (((int)(Math.random()*10))% 3);
        int nolr = 1 + (((int)(Math.random()*10))% 3);
        int nclients = 1 + (((int)(Math.random()*50)));
        String sid = "SID_"+i;
        s.setService_ID(sid);
        fault = s.generateFault();
        s.setFault(fault);
        s.setNumberOfReplicas(nor);
        s.setNumberOfLReplicas(nolr);
        s.setClients(nclients);
        gfaults.add(s);
     }
     return gfaults;
    }
 public Faults generateFault()
    {
     int check = 1 + (int)(Math.random()*100);

        Faults f = new Faults();
        f.setFault_Id("STF");
        f.setCheckpointDataSize(check);
        faults.add(f);
        Faults f1 = new Faults();
        f1.setFault_Id("SOF");
         f1.setCheckpointDataSize(check);
        faults.add(f1);
        Faults f2 = new Faults();
       f2.setFault_Id("SUF");
        f2.setCheckpointDataSize(check);
        faults.add(f2);
         Faults f3 = new Faults();
        f3.setFault_Id("SIF");
         f3.setCheckpointDataSize(check);
         faults.add(f3);
        Faults ff = null;
        int pos = ((int)(Math.random()* 10 ))% 4;
```

```
        ff = (Faults)faults.get(pos);

     return ff;
    }
    public int getNumberOfReplicas() {
        return numberOfReplicas;
    }
    public void setNumberOfReplicas(int numberOfReplicas) {
        this.numberOfReplicas = numberOfReplicas;
    }
    public int getClients() {
        return clients;
    }
    public void setClients(int clients) {
        this.clients = clients;
    }
    public int getNumberOfLReplicas() {
        return numberOfLReplicas;
    }
    public void setNumberOfLReplicas(int numberOfLReplicas) {
        this.numberOfLReplicas = numberOfLReplicas;
    }
    public double getReliability()
    {
      int k = (5 +((int)(Math.random()*100)))%20;
      double p = Math.random();
      double reliability1 = Math.pow((1- p),(double)k);
      return reliability1;
    }
    public void setReliability(double reliability) {
        this.reliability = reliability;
    }
    public long getStart_time() {
        return start_time;
    }
    public void setStart_time(long start_time) {
        this.start_time = start_time;
    }
}
```

A.3 Database accessing Implementation

```
package faultrecovery;
import java.sql.*;
import java.util.ArrayList;
/**
 *
 * @author Sihle
 */
public class ServiceDB {
Statement stat = null;
    ResultSet result = null;
    Connection con  =null;
    Time t = new Time();
     public ServiceDB() {
        initialize();
    }
  public void initialize(){
try{
          Class.forName("com.mysql.jdbc.Driver").newInstance();
           con = DriverManager.getConnection ("jdbc:mysql://localhost/services",
"root", "sotobe");
           stat = con.createStatement();


   }
    catch (Exception e)
   {
        e.printStackTrace();
   }
}
   public ArrayList retrieveReplicas(String sid)
{
     ArrayList record = new ArrayList();

try{

          String query = "Select * from replica where sid = '"+sid+"';";
           result = stat.executeQuery(query);
         while(result.next())
         {
           Replicas r = new Replicas();
           r.setSid(result.getString("sid"));
           r.setRid(result.getString("rid"));
           r.setCapacity(result.getInt("capacity"));
           r.setDemand(result.getInt("demand"));
           r.setStart_Time(t.getTime());
           r.setReliability(result.getDouble("s_safty"));
           record.add(r);

              }

   }
    catch (Exception e)
   {
        e.printStackTrace();
   }
     return record;
}
 public void insertService(String sid,int replicas, int restime)
   {
    String query = "insert into service values('"+sid+"',"+replicas+","+restime+");";
   try{

           stat.executeUpdate(query);

   }
    catch (Exception e)
   {
        e.printStackTrace();
   }

   }
 public void insertReplica(String sid,String rid,int capacity, int demand,double
s_safty)
   {
    String query = "insert into replica
values('"+sid+"','"+rid+"',"+capacity+","+demand+","+s_safty+");";
   try{
           stat.executeUpdate(query);

   }
    catch (Exception e)
   {
        e.printStackTrace();
   }

   }
}
```