Integrating Adaptive Services into Grid Resource Brokering

by

Abdulla Othman

Submitted in accordance with the requirements for the degree of
Doctoral of Philosophy

The University of Leeds
School of Computing

March, 2006

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Abstract

Grid computing is highly dynamic in nature where resources are subject to change due to performance degradation and node failure. The resources include processing elements, storage, network, and so on; they come from the interconnection of parallel machines, clusters, or any workstation. One of the main properties of these resources is to have changing characteristics even during the execution of an application. Thus, resource usage by applications cannot be static during run-time; neither can change in resources be considered as faults. Therefore, Grid application designers must keep in mind that resources and resource management are highly dynamic within Grid architectures.

Grid resource brokering is introduced to simplify resource discovery, selection, and job submission for Grid application. However, it is the responsibility of a Grid resource broker to distribute jobs among heterogeneous resources and optimise the resource usage. As a result, a Grid resource broker should have the capability to adapt to these changes and take appropriate actions to improve performance of various computing applications. To adapt to the Grid resource changes, an adaptive service is introduced in this research. The adaptive service consists of a monitoring tool, decision manager, and migration engine to ensure the job finishes at the time specified. The adaptive service supports job migration during run-time to ensure timely job completion.

Our work in this research shows a Grid test-bed and White Rose Grid implementation of an adaptive service that supports job migration during run-time to ensure timely job completion. Performance prediction is used to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user specified deadline. A key feature of our approach is that the user is not required to install additional software, or make complex alterations to their code requiring specialist Grid computing knowledge. This is achieved using a reflective technique to bind the adaptive service components to the user's code. Also, this research proves the adaptive service overhead is very minimal. The adaptive service is a viable contender for future Grid resource brokering implementation.

Memory

I dedicate this thesis in the memory of my father who died in the end of October 2005 when I started writing up. May Allah have mercy upon his soul and may he grant him the highest above in Jannat (Paradise). He was my biggest support in this life. I lost very close friend.

Acknowledgment

At the end of my thesis I would like to thank all those people who made this thesis possible and an enjoyable experience for me.

Firstly, I am deeply indebted to my supervisors Karim Djemame and Peter Dew, whom during this period have not only kept me on track, but provided invaluable advice and support. It was an honour working with both of you and having you as supervisors.

Secondly, during my study I would like to thank several people. First, my wonderful family for the daily supports and encouragements. Second I would also like to thank Roger Boyle for his great help during my writing up. Roger allows me to use private office to help me concentrate in my writing. Third, I must also mention Judi Drew for her secretarial help. Finally, my research group specially Duncan Russell for his technical help and advice during my research.

Finally, a very big thanks to two great people who without them this work would be harder to achieve, they are Iain Gourlay and John Hodrien. Thanks for your help and support every time I struggled. I'm so grateful for all the times we discussed ideas, my sincere appreciation for the lovely enjoyable times.

For all my beloved and even those whom I have not acknowledged, may God bless you and grant you with all the success and happiness you desire.

Declaration

List of papers that have been published during the work in this research

1. Djemame, K., I. Gourlay, M. Haji, A. Othman, and J. Padgett. *Resource Management on the White Rose Grid: Current Research Efforts*. in *20th UK Performance Engineering Workshop (UKPEW'2004)*. 2004. Bradford, UK.
2. Othman, A., K. Djemame, and I. Gourlay, *A Monitoring and Prediction Tool for Time-Constraint Grid Application*. Lecture Notes in Computer Science. 2004. 66.
3. Othman, A., P. Dew, K. Djemamem, and I. Gourlay. *Adaptive grid resource brokering*. 2003.Cluster2003. Hong Kong, China: IEEE Comput. Soc.
4. Othman, A., P. Dew, K. Djemame, and I. Gourlay. *Toward an Interactive Grid Adaptive Resource Broker*. in *UK e-Science All Hands Meeting*. 2003. Nottingham, UK.

Some works are been interested on my research and referenced it.

1. Pathak, j., j. Treadwell, r. Kumar, P. Vitale, and F. Fraticelli, *A Framework for Dynamic Resource Management on the Grid*, http://www.hpl.hp.com/techreports/2005/HPL-2005-153.pdf, Editor. 2005, Hewlett-Packard Laboratories & Office of Corporate Strategy and Technology, Palo Alto, CA: USA.
2. Champrasert, P., T. Itao, and J. Suzuki. *Symbioticsphere: a biologically-inspired network architecture for autonomic grid computing*. 2005.
3. Champrasert, P., C. Lee, and J. Suzuki. *SymbioticSphere: An Architecture for Autonomic and Emergent Networking*. in *the 2nd IEEE Upstate New York Workshop on Communications and Networking*. 2005. Rochester, NY.
4. Tramontana, E. and I. Welch. *Reflections on Programming with Grid Toolkits*. in *ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*. 2004. Oslo, Norway

5. Haji, M.H., P.M. Dew, K. Djemame, and I. Gourlay. *A SNAP-based community resource broker using a three-phase commit protocol*. 2004.

Contents

List of Figures

List of Equations

List of Tables

Glossary

$T_{100\%}$: The time required for an application to complete execution (from the beginning) on a specified resource at 100% CPU usage, i.e. the CPU time required to complete its execution.

$T_{100\%}^{M}$: The time required for a partially completed application to complete execution on a resource it is migrated to, from the time it begins executing on the new resource at 100% CPU usage, i.e. the remaining CPU time required on the new resource to complete its execution.

$T_{CPU}(t_0, t_i)$: The CPU time dedicated to an application on a specified resource between the times t_0 and t_i.

T_c: Used in the thesis to represent application completion time (see equation 4.6 and the definition of $\overline{F}(t_i, T_c + t_i)$).

$\overline{F}(t_i, T_c + t_i)$: Mean CPU usage between the times t_i and $T_c + t_i$. Used in equation 4.6 to represent the predicted mean CPU usage from time t_i until an application completes execution.

$T_{totalCPU}$: The total CPU time spent on an application since it began execution.

$T_{totalWall}$: The total wall time elapsed since an application began execution.

$\overline{CPU}\%$: Mean CPU usage of an application since it began execution.

$CPU_{current\%}$: The CPU usage is monitored at regular time intervals. This represents the CPU usage over such a time interval.

$T_{CPU\ remaining}$: The remaining CPU time required to complete execution of an application on the resource it began executing on.

$T_{remaining}$: Predicted remaining wall time for an application to complete execution on the resource it began executing on.

$T_{remaining(NEW)}$: Predicted remaining wall time for an application to complete execution if it is migrated to an alternative compute resource.

$T_{CPUremaining(NEW)}$: The remaining CPU time required to complete execution of an application if it is migrated to an alternative compute resource.

$T_{elapsed}$: Time elapsed since an application began execution.

T_{user}: User-specified time that is the maximum amount of time the job is allowed to take in order to meet the user’s requirements.

$\overline{T}_{transfer}$: Time it takes, on average, to transfer files needed by an application from the resource it is executing to the resource that is under consideration for job migration.

$\overline{T}_q^M$: Mean queuing time for the application on the resource that is under consideration for job migration.

in-fra-struc-ture \'in-fre-,strek-cher, n (1927)
The basic facilities, services, and installations needed for the functioning of a community or society, such as transportation and communications systems, water and power lines, and public institutions including schools, post offices, and prisons.
— American Heritage Dictionary

Chapter 1

Introduction

1.1 Introduction

The construction of Roman roads, the telegraph, the telephone, the modern banking system, the railway, the motorway, the electrical power Grids, and the Internet, are all successful infrastructures that have revolutionised how people communicate and interact. At the end of the last century, we witnessed the birth of what promises to be the next revolutionary infrastructure. Funded in the USA by several governmental agencies, including the National Science Foundation (NSF), the Defence Advanced Research Project Agency (DARPA), the Department of Energy (DOE), and the National Aeronautics and Space Administration (NASA), in the UK the Core e-Science programme was funded by both the Office of Science and Technology (OST) and Department of Trade and Industry (DTI), and was managed by The Engineering and Physical Sciences Research Council (EPSRC) on behalf of all the Research Councils, this new infrastructure is often referred to as Grid computing [5-8]

Grid computing is a specialised instance of a distributed system [9, 10] with the following characteristics: computer and data resources are geographically distributed; they are under the control of different administrative domains with different security and accounting policies; and the hardware resource base is heterogeneous, consisting of PCs, workstations, Network, and supercomputers from different manufacturers.

The Grid computing discipline involves the actual networking services and connections of a potentially unlimited number of ubiquitous computing devices within a Grid. This new innovative approach to computing can be most simply thought of as a massively large power utility Grid, such as the power provided to our homes and businesses each and every day [11]. This delivery of utility-based power has become second nature to many of us, worldwide. We know that by simply walking into a room and turning on the lights, the power will be directed to the proper devices of our choice for that moment in time. In this same utility fashion, Grid computing openly seeks and is capable of adding an infinite number of computing devices into any Grid environment, adding to the computing capability and problem resolution tasks within the operational Grid environment.

1.2 Research Motivation

Grid computing [11] is highly dynamic in nature where resources are subject to change due to performance degradation and node failure. The resources include processing elements, storage, network, and so on; they come from the interconnection of parallel machines, clusters, or any workstation. One of the main properties of these resources is to have changing characteristics even during the execution of an application. Thus, resource usage by applications cannot be static during run-time; neither can change in resources be considered as faults. Therefore, Grid application designers must keep in mind that resources and resource management are highly dynamic within Grid architectures.

On the other hand, writing a Grid application is at least as difficult as writing an application for traditional distributed systems due to the complexity of Grid environments. Thus, programmers must deal with issues of application distribution, communication and synchronisation. Furthermore, the Grid presents additional challenges, as programmers may be required to deal with issues such as security, disjoint file systems, fault tolerance and adaptivity, to name only a few [7, 12]. Without additional higher level abstractions, all but the best programmers will be overwhelmed by the complexity of the environment.

1.2.1 Need for Adaptivity in Grid Brokering

Grid resource brokering is introduced to simplify resource discovery, selection, and job submission for Grid application. Resource broker is defined as a middleware component that makes scheduling decisions involving resources over multiple administrative domains [13]. A resource broker supports transparent access to resources and consequently transparent application execution. However, it is the responsibility of Grid resource broker to distribute jobs among heterogeneous resources and optimise the resource usage. As a result, a Grid resource broker should have the capability to adapt to these changes and take appropriate actions to improve performance of various computing applications. For example, if a node running an application crashes, then the resource broker should migrate the jobs running on that machine to an alternate one. To adapt to the Grid resource changes, an adaptive service is introduced in this research.

The adaptive service consists of a monitoring tool, decision manager, and migration engine to ensure the job's finishing at the time specified. The adaptive service supports job migration during run-time to ensure timely job completion. Performance prediction is used to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user specified deadline.

1.2.2 Motivating Scenario

The following scenario is intended to illustrate the importance of resource brokering and support for adaptation.

The DAME (Distributed Aircraft Maintenance Environment) project [14] addressed the research problem of developing a Grid-based diagnostic environment for aircraft engines. The DAME system consists of a set of Grid applications (e.g. vibrational data analysis, data pattern matching) that can be used to support engine diagnosis. Hence the DAME system is designed to increase reliability for passengers and profit margins for the aircraft operator (by minimising schedule disruption and unscheduled maintenance). A typical scenario in this context is as follows: when an aircraft lands, the engine must go through a series of inspections/checks before it can be released. The individual responsible for this decision (the Engine Releaser) requires the assistance of a maintenance team, who have access to tools that comprise the system. In order to benefit from the use of the DAME system, the maintenance team must receive results within a deadline in order to minimise schedule disruptions and ensure any required actions (e.g. maintenance work) can be scheduled in a timely fashion. In order to achieve this,

- Support for transparent access to resources and job submission is necessary.
    - Note that the users of the DAME system (the maintenance team) are not computer experts and need support for job submission without the need to understand Grid middleware complexities.
- A means of addressing performance degradation during run-time is highly desirable.
    - If a DAME application is executing slower than expected, e.g. due to another application sharing the same CPU or a problem with the compute resource, then without run-time adaptation, results may not be obtained within the specified deadline. Hence support for job migration (to alternative resources) is needed to address this problem.

1.3 Research Context

Adaptivity in the resource broker needs to cater for the user's requirements, ensuring a) appropriate resources are selected that have the capability to meet their requirements prior to run-time and b) resources are fulfilling user requirements during run-time. A framework that fits well for this form of brokering is Adaptive Grid Resource Brokering [15], as it is inspired by the user requirements to reconcile the needs of the user with those of the resource providers. The user requirements are examined, and resource providers that can support such requirements are identified to cater for the job execution.

This research proved that the framework developed enables the easy integration of adaptive service into the Grid application. To support this, we developed a prediction model and mapped it to our adaptive service. This prediction model is likely to be used in Grid applications. We incorporated this model to off-the-shelf applications. We used the framework with applications written in C and Java programming languages without the need to alter the code.

Thus the reflective technique[16] is used in this research for the binding between the adaptive service and user application. The adaptive service provides an enhancement over the traditional resource broker in terms of the time taken between submission (to the broker) of user requirements and the job-finishing execution.

This thesis evaluates the adaptive service which includes monitoring, decision-making, prediction model, and migration. The evaluation involves the use of a Grid test-bed and the White Rose Grid (WRG) [17]. The evaluation also includes studying the overhead which occurs when adaptive service is used.

1.4 Aims and Objectives

The first aim of this research is to develop an adaptive service which ensures the user’s jobs finish at the time specified. The other aim is to integrate the adaptive service in the resource broker in the context of computational Grids. This is to insulate the users from the complexities of Grid middleware and environments, alleviating them from the burden of having to know the various mechanisms of the Grid and the environment behaviour. The research would apply a computational-intensive application and the DAME XTO (eXtract Tracked Orders) application as an exemplar for the need to fulfil the user’s requirements. Both have a time limit and CPU power requirements to run in the Grid. The computational-intensive application is a random batch job which consumes CPU power to the maximum. The DAME project designs and implements a prototype system to facilitate the diagnosis and maintenance of aircraft engines through Grid computing. This is motivated by the need to reduce the cost of unexpected and unplanned maintenance of aircraft engines by processing and diagnosing the problems as they occur, which is why there is a need to ensure the jobs finish at the time required at any Grid resource.

With this in mind the objectives of the research are:

- To develop adaptive service to fulfil job requirements at particular resource. The adaptive service includes a monitoring tool, decision manager, and migration engine to assist achieving its goal. The adaptive service evaluates and makes decisions when the user requirements are violated.

- To make the adaptive service transparent to the user, this insulates the user from the Grid environment complexity. The user is not required to alter code, or have knowledge about the Grid infrastructure. The current adaptive brokering systems have not been tailored to adapt without the user changing code, thus the user is required to have considerable knowledge of the Grid.
- To integrate adaptive service in the resource broker, which distribute jobs among heterogeneous resources and optimises the resource usage. Combining the resource selector and estimator in resource broker prior to run-time with the adaptive service during run-time to achieve the optimal resource performance.
- To run the adaptive service without the need to install new software: build adaptive services to run in any resource without the need to have special software installed. This will enable the adaptive service to run on any resource in the Grid.

1.5 Properties of the Adaptive Service

Our goal is to simplify the construction of adaptive Grid applications. We believe a good solution for achieving this goal should exhibit the following properties:

- Separation of concerns and composition. Designing and writing adaptive code are complex and error-prone tasks in a Grid environment for application programmers or tools developers, as the application will be running on random machines with different environments. Furthermore, the incorporation of an adaptive service should not interfere with other non-functional concern such as security.
- Localised adaptive service. This means that the adaptive service will be running at the same resource as the user’s job. This reduces network overhead and increases efficiency. The monitoring data is processed locally without network delay since the purpose of the adaptive service is to finish the job within the time required. However, an instance of the broker is created for each user in order to communicate with the adaptive service during run-time in case of resource degrading.

- Working proof of concept. This research should be able to demonstrate the ability to prove the integration of adaptive services in a resource broker using multiple applications written in different programming languages. Further, applications with an integrated adaptive service should be able to tolerate more failures than applications that do not use any adaptive service.

1.6 Evaluation

Based on the aims and objectives and the properties listed in 1.4 and 1.5, we have derived several criteria by which to evaluate the framework:

- Does the use of the adaptive service result in shorter job execution time, compared to the case when the adaptive service is not used?
- In the case of resource degradation, when job requirements are not met, are jobs being successfully migrated?
- What is the impact on execution time of sharing CPU between the application and the adaptive service? Is the overhead of using adaptive service due to the prediction algorithm or to the framework itself?

1.7 Major Contributions

The contributions of this work are as follows:

- Development of the framework for simplifying the construction of Grid applications. The framework insulates the user from the complexity of Grid middleware and the Grid resource environment.
- Developing an extension to the simple resource broker. The extension includes historical database, which is used by resource selector and estimator components. The resource selector searches the database for historical information about the candidate resources, and the resource with higher probability of failure will be eliminated. The estimator is able to predict the queuing time and file transfer time.

- Development of the adaptive service to ensure the user's requirements has been fulfilled during run-time. The adaptive service's functionality is incorporated into the resource broker and consists of (1) monitoring tool, (2) decision management, and (3) migration engine.
- Developing a prediction model for the decision manager to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user specified deadline. The prediction method used here is based on the pattern of CPU usage during run-time, and is therefore expected to display a high level of accuracy only for CPU-intensive applications.
- The use of reflection technique [18-20] to bind the adaptive service with user's application. This extends user application with additional capabilities through composition.

To the best of our knowledge, this is the first research to advocate and use reflective technique in a computational Grid. Moreover, we are the first to demonstrate the adaptive service in Grid applications without altering the user's applications.

1.8 Outline

The rest of the thesis been organised as follows. In Chapter 2, we present an overview of related work in the areas of computational Grids, middleware for executing a Grid job, Grid job management, adaptivity in the Grid and reflection technique. In Chapter 3, we provide a design of our overall model, the adaptive service and resource broker. In Chapter 4, we describe the implementation detail of the adaptive service that include resource selection and historical database in resource broker prior to run-time, monitoring tools, decision manager and migration in the adaptive service during run-time. In Chapter 5, we present the experimental results and discussion about the framework developed. This includes lessons we learned and issues raised during the experiments. In Chapter 6, we conclude by presenting the limitations, future work, and conclusion.

Next, an overview of 1) types of Grid, 2)Grid middleware, 3) job management system which include adaptive Grid job management, 4) monitoring application, 5) Migration, and finally 6) reflective technique.

There is only one nature – the division into science and engineering is a human imposition, not a natural one. Indeed, the division is a human failure; it reflects our limited capacity to comprehend the whole.
— Bill Wulf

Chapter 2

Literature Review

2.1 Introduction

Grid computing refers to the integration of computing resources for solving large-scale and complex problems. Although Grid computing has been around for some time, particularly in the academic, military and research communities, its popularity has fast increased in recent years due to the availability of new standards and other technological infrastructures. As an emerging technology, Grid computing is still characterised by diverse definitions and descriptions. It is probably best to describe it along the same lines as we describe the well known power Grid. It could be a machine, network, or some service that is synthesised using a combination of machines, networks, and software. The resource provider is defined as an agent that controls the resource. For example, a resource broker that acts as the resource provider for a resource could provide the consumers with a ‘value added’ abstract resource [3].

However, a computational Grid, by nature, is a collection of global resources which are decentralised and loosely coupled [7]. They span across multiple administrative domains and geographical boundaries with no absolute central full control of the resources. This introduces several challenges[21] that underlay the construction of computational Grid, which are listed below:

- *Site autonomy,* means resources are owned and operated by different organisations which may have different policies, security mechanisms etc. The owner of these resources has total control over their usage.
- *Heterogeneous substrate,* Different sites may use different local resource management systems like Condor [22], Sun Grid Engine (SGE) [23], and Portable Batch System (PBS) [24] etc. Even if two different sites use the same local resource management system they might differ in configuration leading to difference in functionality.
- *Policy extensibility,* as resources on the Grid are drawn from a wide range of domains, each with its own requirements and configurations. Resource Management should support frequent changes of domain specific management structures without any requirement to change the code, thus supporting policy extensibility. Resource management should adapt itself to new user requirements and should be capable of evolving itself to meet future demands.

This chapter is organised as follows. First the different types of Grid, then the core Grid middleware used in this research to manage the resources. Secondly, the job management system is presented, which includes general job and adaptive job management systems. Thirdly, the approaches used by other research to achieve adaptivity and description of adaptive service components. Finally, the reflective technique used to bind the job with the adaptive service.

2.2 Types of Grid

As defined above, Grid computing is becoming increasingly popular, with applications in many areas. In general, Grids can be categorised in accordance with the type of solutions they seek to provide. Three main Grids are thus distinguished, although typically they largely overlap in scope, and a typical Grid will usually be a combination of two or more of the three. In practice, the Grid environment impinges on many of the decisions made by software developers. The following are the three main types of Grids.

2.2.1 Data Grid

A data Grid [25] refers to a system that is responsible for housing and providing access to data across multiple organisations. In other words, a data Grid computing system is concerned with the control, sharing and management of a vast amount of data. The development of computational Grid systems has obviously contributed to the development in the acquisition and processing of massive amounts of data.

Under the data Grid scenario, users will usually have little or no interest at all in the locations of the data, as long as they are guaranteed a smooth access. Data Grid is probably the fastest growing type of Grid computing and finds applications in virtually any field of study. Consider a number of researchers in three continents, each with a unique set of DNA data. A data Grid computing system makes it possible for them to conveniently share and manage their data. The system should allow to have in place security controls permitting or limiting data access according to pre-set criteria. Examples of practical applications is EU DataGrid [26].

2.2.2 Scavenging Grid

The concept of scavenging Grid is generic, and it is most commonly used to describe a situation in which a vast number of desktops are scanned for whatever computing power is available, typically in the form of CPU cycles and other resources. Owners of the desktop machines are usually given control over when their resources are available to participate in the Grid. The scavenging is the most commonly used Grid. A more detailed discussion is in [1].

2.2.3 Computational Grid

As the name suggests, the focal point of this Grid is on computational power. Basically, the Grid involves setting aside resources specifically for ensuring computing power, which usually involves high performance servers. Although the concept of computational Grid was originally used to refer to hardware and software pooling together resources from different computing sources to solve a single problem, it has acquired a broader interpretation in recent years. In particular, it is now increasingly used in a broader sense to refer to any hardware and software system used in co-ordinating shared resources within any dynamic entity consisting of individuals, institutions, and other resources [27, 28].

The computational Grid is made up of a number of components from enabling resources to end user applications. A layered architecture of a computational Grid is shown in Figure 2.1, discussed in greater depth in [11]. At the bottom of the Grid stack are distributed resources managed by a local resource manager with local policy, and interconnected through local or wide area networks. The layers can be described as follow:

- *Grid fabric*, this consists of all the globally distributed resources that are accessible from anywhere on the Internet. These resources could be computers (such as PCs, clusters, or parallel computers) running a variety of operating systems (such as UNIX or Windows), storage devices, databases, and special scientific instruments such as a radio telescope or particular heat sensor.
- *Security infrastructure*, A secure and authorized access to Grid resources is provided by *security infrastructure*. The layer on top is called *Core Grid middleware*. This offers core services such as remote process management, co-allocation of resources, storage access, information registration and discovery, security, and aspects of Quality of Service (QoS) such as resource reservation and trading.
- *User-level Grid middleware,* includes application development environments, programming tools and resource brokers for managing resources and scheduling application tasks for execution on global resources.
- *Applications and portals,* Grid applications are typically developed using Grid-enabled languages and utilities such as Java CoG, High Performance C++ (HPC++) or Message Passing Interface (MPI). An example application, such as parameter simulation or a grand-challenge problem, would require computational power, access to remote data sets, and may need to interact with scientific instruments. Grid portals offer Web-enabled application services, where users can submit and collect results for their jobs on remote resources through the Web.

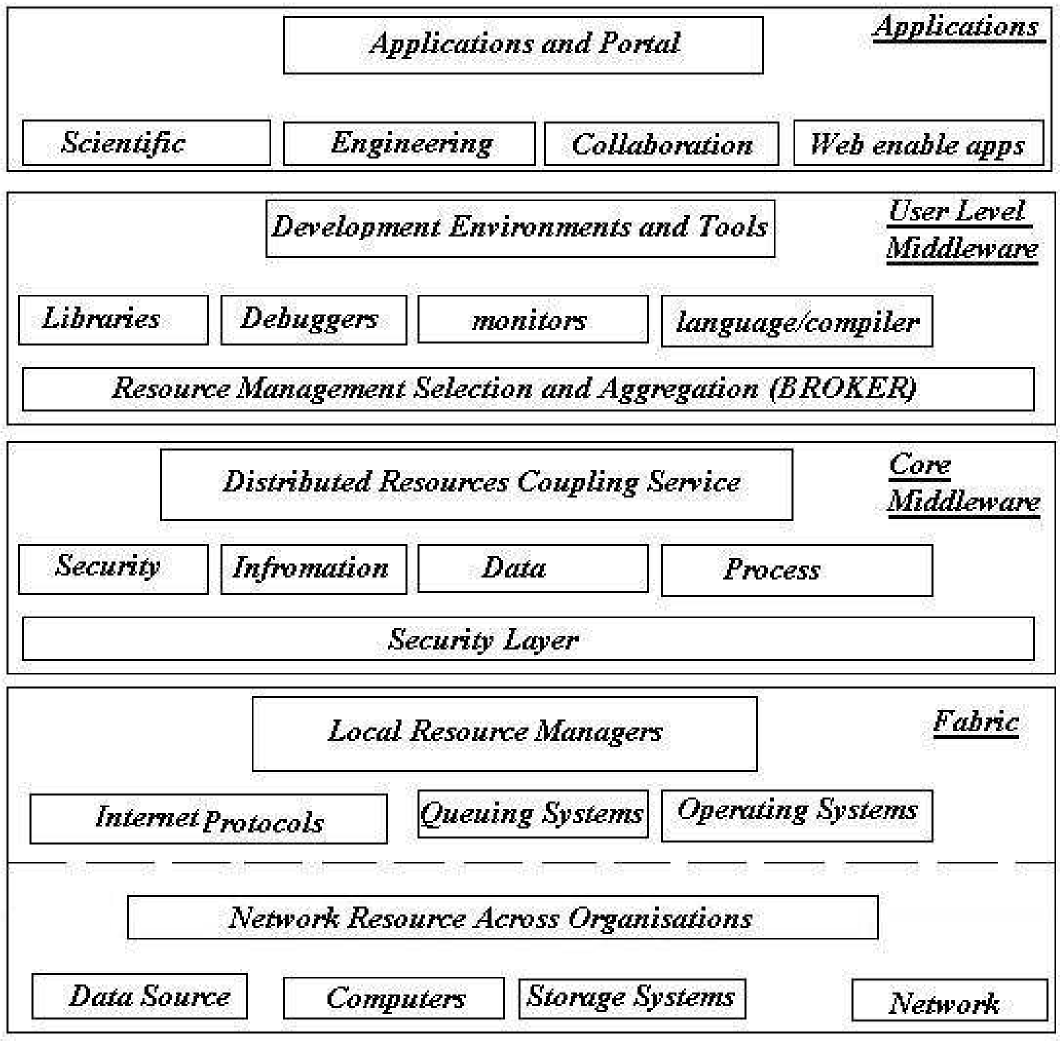

Figure 2.1 : A layered architecture of computational Grid and technologies (Adapted from[7])

The upshot is that application developers in need of greater or optimal performance may need to utilise available resources more effectively. To do that, the developers have to leverage Grid technology and Grid-enable those applications. This can be achieved in many different ways [29]. [30] provides a detailed discussion on how people and machines interact to effectively capture, publish, share and manage knowledge resources.

In order to utilise the Grid, a user must have the ability to interact with the Grid middleware such as the ones predominantly used in the Grid community, namely Globus or Sun Grid Engine. Those middleware are used in this research to discover resources, select resource, and submit the jobs. More of those middleware are discussed in the next section

2.3 Middleware for Executing an Application on the Grid

This section describes the technologies which are used in this research as the core Grid middleware layer (discussed in Section 2.2.3). This middleware is the link between the resources and the outside world (e.g. broker, scheduler, or even user).

2.3.1 Globus Toolkit

Current networking technologies are not performance-driven and are limited to old fashioned client-server models, also they focus on supporting communication rather then computation. The most prominent example is the Internet. The current model of distributed computing requires to integrate remote resources into a computational common place and build Grid specific middleware that addresses the needs of computations, including dynamic resource allocation, resource co-allocation, heterogeneity and dynamic computation, as well as communication substrates and process-oriented security[31]. The Globus Toolkit (GT) [32] project is one of the first steps in this direction. The GT project is a multi-institutional research effort to provide persistent, dependable and reliable access to geographically distributed Grid resources. One of its main goals is to understand application requirements for a usable Grid and to develop the essential techniques to meet these requirements.

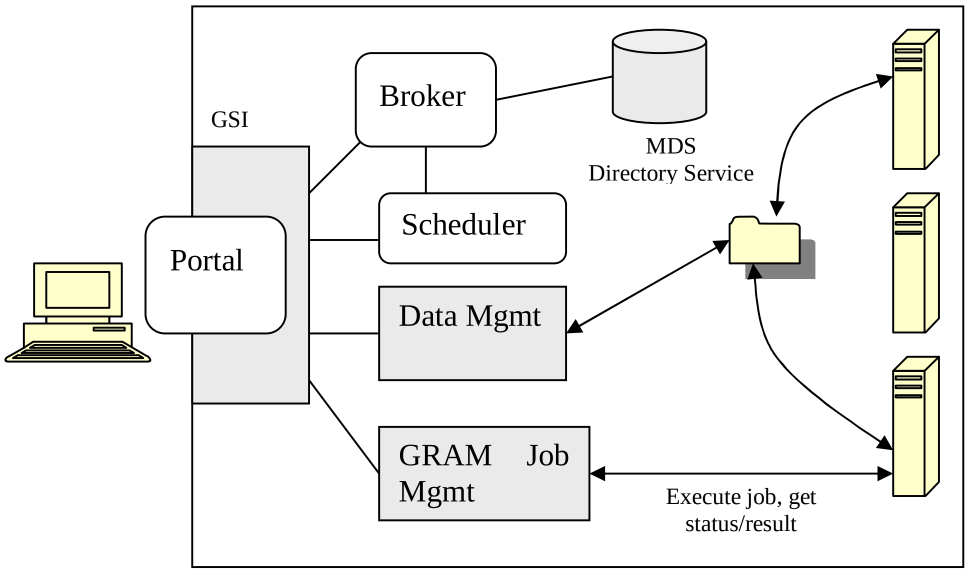

Figure 2.2: Globus Toolkits Architecture (adopted from [1])

The GT provides a set of basic facilities (depicted in Figure 2.2) needed for Grid computing such as security, execution manager, data management and information service, which are discussed in turn.

2.3.1.1 Security

A major requirement for Grid computing is security. At the base of any Grid environment, there must be mechanisms to provide security, including authentication, authorization, data encryption, and so on. The Grid Security Infrastructure (GSI) [33] component of the GT provides robust security mechanisms, and includes an OpenSSL implementation. It also provides a single sign-on mechanism, so that once a user is authenticated, a proxy certificate is created and used when performing actions within the Grid.

GSI is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol. Extensions to these standards have been added for single sign-on and delegation. The GT's implementation of the GSI adheres to the Generic Security Service API (GSS-API), which is a standard API for security systems promoted by the Internet Engineering Task Force (IETF) [34].

A central concept in GSI authentication is the certificate. Every user and service on the Grid is identified by means of a certificate, which contains information vital to identifying and authenticating the user or service. A GSI certificate includes four primary pieces of information:

- A subject name, which identifies the person or object that the certificate represents.
- The public key belonging to the subject.
- The identity of a Certificate Authority (CA) that has signed the certificate to certify that the public key and the identity both belong to the subject.
- The digital signature of the named CA.
- GSI certificates are encoded in the X.509 certificate format, a standard data format for certificates established by the Internet Engineering Task Force (IETF).

2.3.1.2 Execution Management

This is the core of Globus Toolkit (GT) services. The GT Resource Allocation Manager (GRAM) is a basic library service that provides capabilities to do remote-submission job start-up. GRAM unites Grid machines, providing a common user interface so that a user can submit a job to multiple machines on the Grid fabric.

GRAM processes the requests for resources for remote application execution, allocates the required resources, and manages the active jobs. It also returns updated information regarding the capabilities and availability of the computing resources to the Monitor and Discovery System (MDS) [35].

GRAM provides an API for submitting and cancelling a job request, as well as checking the status of a submitted job. GRAM is the lowest level of Globus resource management architecture.

GRAM is responsible for parsing and processing RSL specifications that outline job requests. The request specifies resource selection, job process creation, and job control. This is accomplished by either denying the request, or creating one or more processes (jobs) to satisfy the request, enabling remote monitoring and managing of jobs already created. RSL is a structured language by which resource requirements and parameters can be outlined by a user. To run a job remotely, a GRAM gatekeeper (server) must be running on a remote computer, listening at a port, and the application needs to be compiled on that remote machine. The execution begins when a GRAM user application runs on the local machine, sending a job request to the remote computer.

GRAM allows a user to run jobs remotely, providing an API for submitting, monitoring, and terminating the job. When a job is submitted, the request is sent to the gatekeeper of the remote computer. The gatekeeper handles the request and creates a job manager for the job. The job manager starts and monitors the remote program, communicating state changes back to the user on the local machine. When the remote application terminates, normally or fails, the job manager terminates as well.

2.3.1.3 Data Management

Moving data to various nodes within the Grid needs secure and reliable methods. The GT contains a data management component that provides such services. This component, known as GridFTP [36], is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. It is built on top of the standard FTP protocol, but adds additional functions and utilises the GSI for user authentication and authorisation. Therefore, once a user has an authenticated proxy certificate, he/she can use the GridFTP facility to move files without having to go through a login process to every node involved. This facility provides third-party file transfer so that one node can initiate a file transfer between two other nodes.

2.3.1.4 Information Services

Grid resources require a mechanism for publishing and discovering their status and configuration information such as the Monitoring and Discovery System (MDS). MDS [35, 37] simplifies Grid information services by providing a single standard interface and schema for the many information services that are used within.

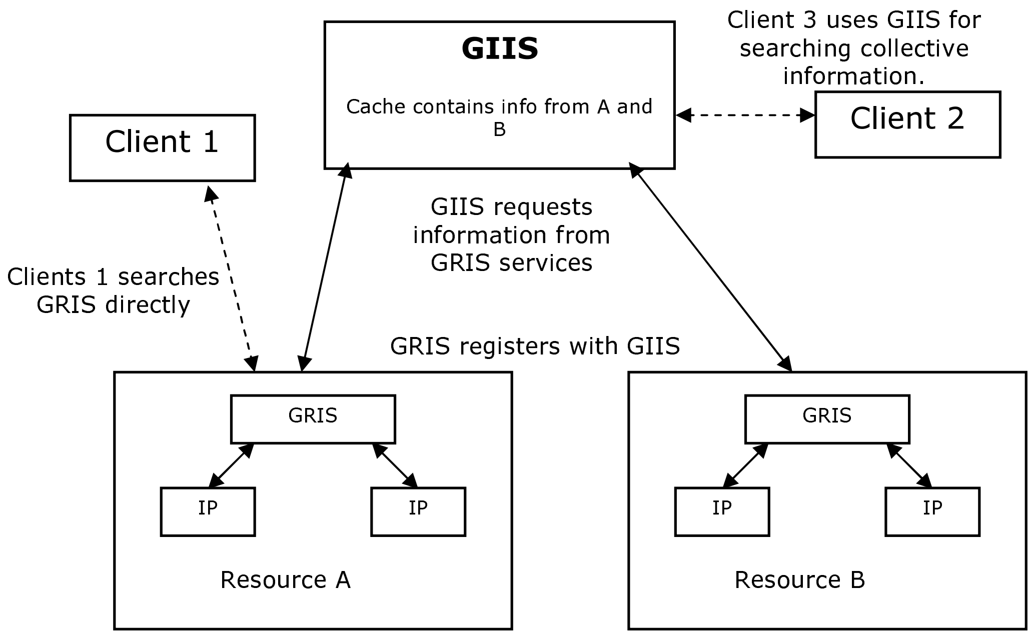
GIIS
Cache contains info from A and B
Client 3 uses GIIS for searching collective information.
Client 1
Client 2
GIIS requests information from GRIS services
Clients 1 searches GRIS directly
GRIS registers with GIIS
GRIS
IP
IP
Resource A
GRIS
IP
IP
Resource B

Figure 2.3: Information service architecture.(Adopted from [3])

The MDS, as shown in Figure 2.3, consists of three main components which are:

- Information Provider (IP) is an interface for any data collection service that gathers information about a particular aspect of a resource such as disk capacity, RAM memory and CPU load. This information is then passed onto the next component that deposits and displays the information as entries.
- Resource Information Service (GRIS), is a distributed information service that can answer queries about a particular resource by directing the query to the underlining IP.
- Grid Index Information Service (GIIS), combines arbitrary GRIS services to offer a coherent system image that can be explored or searched by a Grid client. It provides the mechanism for identifying resources of a particular interest. For example it could list all the computational resources available within a particular research consortium.

The Globus MDS uses the Lightweight Directory Access Protocol (LDAP)[38] as its access protocol, and LDAP object classes as its data representation, but it adopts innovative approaches to the problems of resource registration and discovery. Information about an individual resource or set of resources is collected and maintained by a Grid Resource Information Service (GRIS) daemon, which responds to LDAP requests with dynamically generated information, and can be configured to register with one or more Grid Index Information Services (GIISs). Users will typically direct broad queries to GIIS to discover resources, and then drill down with direct queries to GRIS to get up-to-date, detailed information about individual resources. Information in LDAP is organized in a tree structure referred to as a Directory Information Tree (DIT), with nodes in a DIT tree corresponding to the LDAP structured data types called object classes.

The MDS is widely used in the Grid community such as in many e-science projects and the work mentioned in this research. However, default installation provides limited dynamic information such as a lack of local resource management queuing details. Nevertheless it allows for such information to be integrated into its system.

Submitting a job to Grid middleware can be either through Globus or a local scheduler (e.g. Sun Grid Engine). To understand more about job submission, the last two parts of this section will explain the Sun Grid Engine and the process of job submission.

2.3.2 Sun Grid Engine

Sun™ ONE Grid Engine (SGE)[39] is an example of a Distributed Resource Manager (DRM). SGE is an advanced resource management tool for Grid computing environments. The main purpose is to control to best achieve the shared resources such as productivity, timeliness, and level-of-service. SGE provides advanced resource management and policy administration for UNIX environments that are composed of multiple shared resources. Moreover, SGE provides users with the means to submit computationally demanding tasks to the Grid. Users can submit batch jobs, interactive jobs, and parallel jobs. To illustrate, these jobs correspond to bank customers. The jobs wait in the queue instead of a bank lobby. A queue acts as bank employees, which provides services for jobs. A queue is a container for a class of jobs that are allowed to run on one or more hosts concurrently. The jobs will be associated with the queue, which means that if the queue is suspended then all the jobs associated with that queue will be also suspended. For jobs not submitted directly to the queue, however, the user must specify the job requirements, such as memory, execution speed, available software, and similar needs. The SGE will send out the job to a suitable queue and suitable host with minimum execution loads. This may result in longer job execution time if the load on the machine is high.

SGE-enabled hosts can be master hosts, execution hosts, submission hosts, and administration hosts. These roles are not mutually exclusive; it is possible for a host to perform all four functions. A typical cluster configuration is to have one master host, running the `sge_qmaster` (manager) and `sge_schedd` (scheduler) daemons, and the other hosts running `sge_execd` (execution) daemons. SGE hosts are communicating through TCP/IP; for this purpose, there is a special daemon, `sge_commd`, running on each host[40].

SGE has been integrated with the GT technology in 1999[40], allowing for the globalisation of corporate Grids. SGE has built a strong relationship with Globus, and supports its current activities to establish industry standards related to Grid computing. The integration between SGE and GT is as follows:

- There is an integration of the SGE with GRAM. This means that jobs submitted to Globus, using the Globus RSL, can be passed on to the SGEs. Evidently, the key here is to provide a means of translation between RSL and the language understood by the SGE. These are implemented in Globus using GRAM job manager scripts.
- There is integration with MDS. The use of a GRAM reporter allows information about a SGE to be gathered and published in the MDS. The reporter will run at each campus site periodically via cron (the clock daemon in UNIX that executes commands), and query the local SGE. This means that up-to-date queue information can be gathered across many cluster Grids.

2.3.3 Overview of Job Submission

There are many types of Grid jobs ranging from parameterised, interactive, batch, and MPI to name just few. The user can submit the job to Grid middleware by two different methods.

1. Submit directly to the SGE. This method can be carried out in two different ways. The first, the user can submit his job directly to a selected queue of his choice for execution. This is on the basis that the queue can handle the job which is determined by

the execution run time duration stated in the job’s script. The second is submitting the job directly to the SGE scheduler, but the job is sent to a spool area waiting for the SGE scheduling interval to allocate the job to a queue based on its description.

2. Submit RSL. The first step in this method is to transfer any data files and executables to the remote resource. The second step is to submit an RSL, which provides the details on how to execute the job, which queue to use, and which will also initiate the execution.

Overall, submitting a job through the Grid in relation to a local distributed scheduler, the user needs to have extensive knowledge on how to operate the various Grid technologies. This ranges from querying the information provider to the creation and submission of the RSL. Conversely, submitting the job directly to the local scheduler, the user creates a script describing the job, and either specifies the queue or allows the scheduler to handle it.

There are more projects to execute Grid application and play the role of Grid Middleware such as Legion [41, 42]. Legion is an object-based system designed to harness hosts across multiple sites which are tied together through a high-speed link. Resource management is performed through two specialised objects, an application-specific scheduler and a resource-specific enactor that negotiate with one another to make allocation decisions. Further, it supports a range of applications such as parallel application and parameter sweep studies.

Legion ensures that local policies of participating sites are respected, by allowing the final authority over the use of a resource to be in place. However, it assumes that network resource and protocols currently in use will not change. Also Legion does not support co-allocation. Further, it is written in Mentat Programming Language (MPL) [43], thus it is necessary to have MPL on each platform before Legion can be installed.

2.3.3.1 Resource Specification Language

The Resource Specification Language (RSL)[44] is a language for expressing user resource requests as well as configuration for the application. The basic unit in RSL is an attribute-value pair, which is called a relation. The syntax of the relation is attribute operator value. Examples of operations include ‘=’, ‘!=’, ‘>’ and ‘<=’. The value can either be a single value or a list of values. The syntax of a list is (name_1 value_1) (name_2 value_2) etc. and a resource request can contain one or more relations. There are several methods to construct a job request that contains more than one relation. The most commonly used way to combine relations is the conjunct-request, ‘&’, which is equivalent to logical ‘and’. The disjunct-request ‘|’ can be interpreted as logical ‘or’. Users who request more than one resource should combine the requests by using the multi-request operator ‘+’.

```
& (executable =/bin/program)
  (count = 1)
  (stdin = data.in)
  (stdout = data.out)
```

Figure 2.4: RSL job request specifying executable, input and output files and number of processes.

The RSL example in Figure 2.4 includes four relations combined with the conjunct-request. The attribute executable specifies the program to be run. The standard input and output files for the job are specified with the attributes stdin and stdout. By setting the value of the count attribute to 1, the user requests that one instance of /bin/program should be started on the resource.

```
& (executable = my_app)
  (min_memory > 4096) (max_cpu_time = 120)
  (environment (Home /home/othman)
   (DATADIR /home/Othman/data))
```

Figure 2.5 :RSL job request specifying including a list of values and the greater than operator.

In the example in Figure 2.5, the user requests to run the program my_app. The min_memory attribute specifies that the resource must have at least 4096MB of memory available to the job. Similarly, max_cpu_time is used to determine the maximum execution time of the job. In the example, the user requests the job to run for 120 minutes. The environment attribute is an example of a list of values. The user specifies values for two environment variables, HOME and DATADIR. These variables are then defined before the job starts to execute. Whereas the example in Figure 2.4 only included job configuration, the example in Figure 2.5 contains both configuration, e.g. executable, and requirement on the resource, e.g. min_memory.

2.4 Grid Job Management Systems

This section describes the layer above Globus and local scheduler which tend to hide the complexity of the Grid middleware.

The GT and SGE provide various command line tools for basic job submission and monitoring. The jobs submitted to Grid resources typically run for many hours or even days. It is a cumbersome task for a user to manually log into various resources and submit/monitor jobs over such a long period. Grid job management systems solve this problem by providing a usable interface to access Grid resources. The user submits the job request to a Grid job management system. It searches for suitable resources, submits the job, monitors it on behalf of the user, and finally transfers the output to the user's machine. The user can log into the Grid job management system and check the status of all his/her jobs running on various resources. He/she need not log into each resource individually. Thus the Grid job management system hides the heterogeneity of the Grid from the user, and can be considered as a batch system to the Grid resources. This section examines existing Grid job management systems.

2.4.1 Grid Job Management

Globus is a layered architecture that addresses Grid security [33], remote access and control, providing support for public key infrastructure (PKI) single sign-on authentication/authorisation, an information-rich environment [3] based on the Lightweight Directory Access Protocol (LDAP) and a standardised interface to heterogeneous computing resources [45]. However, the complexity of this middleware hinders the majority of scientists (who are not computer scientists) from doing important work.

The Grid Job Management (GJM) is designed to bridge the gap between those scientists wary of the complexity of the GT and the Grid. In the following, some of the existing GJMs are described.

2.4.1.1 Condor-G

Condor-G [22] provides a computation management agent that resides on the user's machine. The agent is used to submit a job request to the Condor-G scheduler which finds the suitable resource by searching a predefined list of resources. If a suitable resource is not found, the job waits in a persistent job queue and is scheduled later when the resource becomes available. When a resource is available, the scheduler creates a GridManager, which interacts with the Gatekeeper of the resource to perform authentication. Then it stages in the job files using GASS. The Gatekeeper then creates a job manager which submits the job to the local job scheduler. The GridManager periodically queries the job manager to find the state of the job. The GridManager notifies the important changes like failures or job completion to the user through email. The GridManager also manages the proxy credentials of the user or it can be configured to use the MyProxy server. The GridManager as well as the job manager ceases to exist after the execution of a job is completed.

Condor-G handles four kinds of failures: crash of the Globus job manager, crash of the machine that manages the remote resource (the machine that hosts the gatekeeper and job manager), crash of the machine on which the GridManager is executing (or crash of the GridManager alone), and failures in the network connecting the two machines. The GridManager periodically probes the job manager of its job to find the status of the job. If the job manager does not respond, it tries to contact the gatekeeper of the resource. If the gatekeeper also does not respond, then it is assumed that the machine that manages the remote resource has crashed. GridManager waits until the machine comes up and then requests the gatekeeper to start a new job manager. The gatekeeper is automatically started when the machine comes up, as it is usually configured as a system start-up process. If the gatekeeper responds, then only the job manager has failed, and the GridManager requests the gatekeeper to restart the job manager. If there is a network failure, the GridManager waits until it can connect to the job manager, as already described. However, the job manager might have exited if the job has finished execution. The GridManager is unaware of this and thinks that the job manager has failed, so it requests the gatekeeper to restart the job manager. The job manager, when restarted notifies the GridManager of job completion. Local failures of the GridManager itself are handled by checkpointing the GridManager process by using the Condor stand-alone single process checkpointing library.

The fault detection mechanisms make the Condor-G system fault tolerant, but they do not help in detection of application failures. Suppose a parallel job is submitted to a resource but one of the job processes is killed because the machine on which it is running crashes. The other processes of the job that are alive keep waiting for some messages from the failed job and the application hangs.

If the resource management system cannot detect the failure, the job is killed only after it exceeds its time limit, or by human intervention. Condor-G cannot detect that the job has failed inside the resource. Also, Condor-G has no interfaces to support job migration across resources.

2.4.1.2 Nimrod/G

Nimrod/G [46] is a Grid scheduling system developed at Monash University in Australia. The main functionality provided by Nimrod/G is automation of creation and management of large parametric experiments. Besides the plain submission of a request for a resource search, users have an option to specify time and cost constraints which are later used in selecting the resource. If the constraints cannot be met, tradeoffs are explained to the user [46].

It is suggested in [47] that Nimrod worked successfully for static sets of computational resources, but that it exhibited a number of flaws which rendered it unsuitable when “implemented in the large-scale dynamic context of a computational Grid” [47].

- Larger scale Grids in the “real world” typically exhibit a number of different properties:
- Nodes in the Grid are typically scattered across a number of different administrative domains.
- Each domain will typically have its own resource allocation policy, determining for example if resources must be surrendered to local users.
- Each domain will have its own job queuing system, its own access cost, and various amounts of available computational power.

The new system Nimrod/G, which uses the Globus [11] middleware for interfacing with the Grid, has been designed to address these shortcomings. Nimrod/G utilises a Job Wrapper which is responsible for the staging of application tasks and data. The Job

Wrapper acts as a mediator between the parametric engine and the actual machine on which the task runs. Its primary function is to send the results from the remote client back to the main engine. It is therefore conceivable that it could also monitor the resource utilised by the remote client and report those data to the resource accounting system.

Nimrod/G supports an integrated computational economy in its scheduling system. This means that Nimrod/G can schedule jobs on the basis of deadlines and budget but it does not support job migration and adaptation.

2.4.2 Adaptive Grid Job Management

Scheduling long-running applications is a challenging task, since it is difficult to predict resource availabilities into the future. Additionally, Grid systems span multiple administrative domains which may have their own resource policies, viz. local jobs must get precedence over Grid jobs, Grid jobs may only run for a particular length of time, etc. A combination of such policies, dynamic resource availabilities, and similar policies for applications can necessitate migration of jobs from one resource to another during execution. In such a situation, the state of the application can be checkpointed, and the application can be physically migrated to a resource that satisfies both resource and application policies.

Adaptability is defined in [48] as the software which can identify, promote and evaluate new models of code design and run-time support which allow software to modify its own behaviour in order to adapt, at runtime, when exact conditions and inputs are known, to discovered changes in requirements, inputs, and internal/external conditions.

In [49] it is defined as a system that continually (at runtime) monitors its success in achieving its intended goal. When the system is found to be doing poorly, the system modifies itself in an attempt to do better at its assigned task. A self-adaptive system presupposes that there are multiple ways of achieving the same task, and that in some contexts one method may be better suited than another.

In this research, adaptivity defined as software that evaluates user’s job performance, and reacts when the evaluation indicates that the resource is not accomplishing the user’s requirements, or when better functionality or performance is possible.

This implies that the adaptive service has multiple ways of accomplishing its purpose, and has enough knowledge of its construction to make effective changes at runtime. Such services should include functionality for evaluating the resource behaviour and performance, as well as the ability to replan and reconfigure the user's job in order to improve its operation.

Several projects use the word adaptation to describe themselves. As this section presents, all of those projects have different views of what adaptation is, what it consists of and why it should be used. Basically, the only common point is that adaptation consists in changing some things in applications.

Research

2.4.2.1 AppLeS

Application Level Scheduling (AppLeS) focuses on the design, development and deployment of Grid application agents as proposed in [50]. The author note that the computational Grid has spear-headed High Performance Computing (HPC) and hence enhanced potentials for the aggregation of enormous computing resources such as bandwidth, computational power, memory, secondary storage and others, during a single execution. Harnessing this potential is clearly practical in a static (semi-static) and/ or homogeneous computing environment. Things become more complicated as the environment becomes dynamic and heterogeneous.

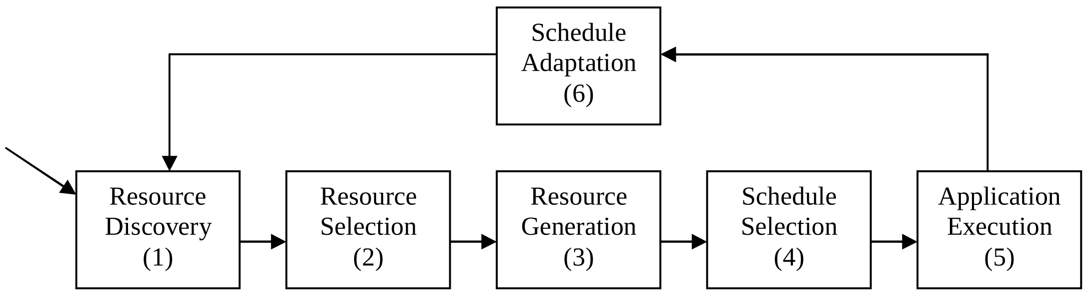

Figure 2.6: Steps of AppLeS Methodology (Adopted from [4])

Thus they propose AppLeS as a methodology for adaptively scheduling and deploying applications in completed dynamic, heterogeneous and multi-user Grid environment. The methodology involves each application being fitted with a customised scheduling agent that monitors resource performance and dynamically generates a schedule for application. Effectively, its mechanics are based on a number of recursive steps (depicted in Figure 2.6), as summarised below:

1. Resource Discovery: Agent must discover the resources that are potentially useful to the application. This can be accomplished using a list of the user’s log-in or by using ambient Grid resource discovery services [3].
2. Resource Selection: Agent typically use an application-specific resource selection model to develop an ordered list of resource sets [51].
3. Resource Generation: Agent applies a performance model to a given list of feasible resource sets, in order to determine a set of candidate schedules for the application on potential target resources.
4. Schedule Selection: Agent chooses the best overall schedule that matches the user’s performance criteria from a given set of candidate schedules.
5. Application Execution : AppLeS agent deploys best schedule on the target resource, using infrastructure available (e.g. Globus [11]).
6. Schedule Adaptation: Agent loops back to Step 1 in case of changing in resource availability. AppLeS targeting long-running applications can then iteratively compute and implement refined schedules.

AppLeS performance prediction methods (on which to base adaptive decisions during run-time) are developed with specific applications in mind. In addition, applications must be AppLeS enabled; which means making alterations to the user’s application code.

2.4.2.2 GrADS

Adaptive execution is also being explored in the context of the Grid Application Development Software (GrADS) project [2]. The aim of the GrADS projects is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. GrADS provides new external services to be accessed by Grid users and, mainly, by application developers to develop Grid-aware applications. Its execution framework [52] is based on three components: the Configurable Object Program, which contains application code and strategies for application mapping; the Resource Selection Model, which provides estimation of the application performance on specific resources; and the Contract Monitor, which performs job interrupting and remapping when performance degradation is detected. GrADS is an ambitious project that involves several outstanding research groups in Grid technology.

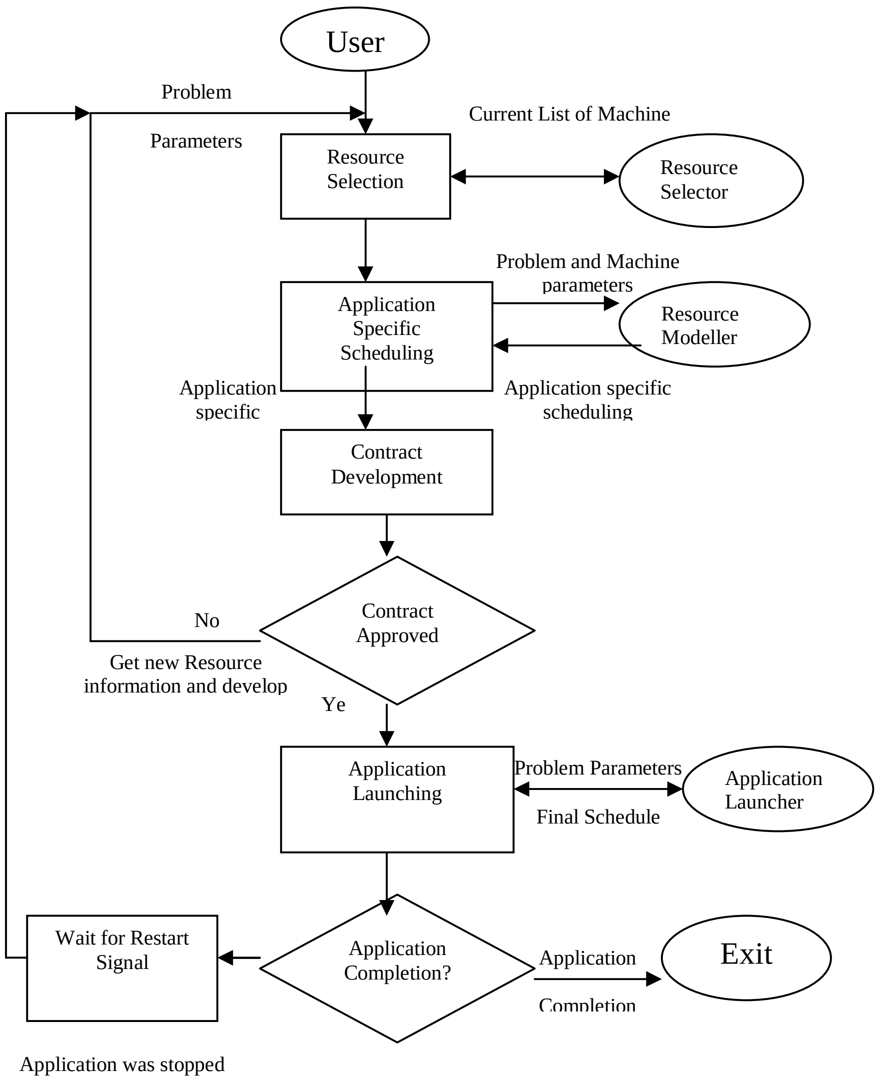

Figure 2.7: GrADS life-cycle(adopted from [2])

The GrADS life-cycle is depicted in Figure 2.7. At a first step, the application manager invokes a component called Resource Selector. This accesses the MDS [35] to get a list of machines that are alive and then contacts the Network Weather Service (NWS) [53] to get system information for the machines. The application manager then invokes a component called Performance Modeller with problem parameters, machines and machine information. The Performance Modeller, using an execution model built specifically for the application, determines the final list of machines for application execution. By employing the application-specific execution model, GrADS follows the AppLeS [2] approach to scheduling. The problem parameters and the final list of machines are passed as a contract to a component called Contract Developer which can either approve or reject the contract. If the contract is rejected, the application manager develops a new contract by starting from the resource selection phase again. If the contract is approved, the application manager passes the problem, its parameters and the final list of machines to Application Launcher. The Application Launcher spawns the job on component called Contract Monitor which through an Autopilot mechanism [54] monitors the times taken for different parts of applications. The GrADS architecture also has a GrADS Information Repository (GIR) that maintains the different states of the application manager and the numerical application. After spawning the application through the Application Launcher, the application manager waits for the job to complete. The job can either complete or suspend its execution due to external intervention. These application states are passed to the application manager through the GIR. If the job has completed, the application manger exits, passing success values to the user. If the application is stopped, the application manager waits for the resume signal, and then collects new machine information by starting from the resource selection phase again.

The GrADS architecture performs adaptivity by adding components called Rescheduler, Migrator and contract Monitor to the GrADS architecture. The contract Monitor monitors the application’s progress, and the Rescheduler decides when to migrate. GrADS implemented a user-level checkpointing library called Stop Restart Software (SRS). The application by making calls to SRS gets the ability to checkpoint data, to be stopped at a particular point in execution, to be restarted later on a different configuration of processors, and to be continued from the previous point of execution.

2.4.2.3 ATLAS

Atlas [55] is a project developed at the University of California at Berkeley. It is designed to execute parallel multithreaded programs on the networked computing resources of the world. The Atlas system is a marriage of existing technologies from Java and Cilck together with some new technologies needed to extend the system into the global domain. The goal of the Atlas system is to exploit the networked resources of the world as a giant distributed computer, and to develop an infrastructure that exploits idle resources both within and among institutions. The idea of adaptation in ATLAS is that the applications must dynamically exploit a varying collection of resources, so that long-running applications can grow, shrink, and migrate as required.

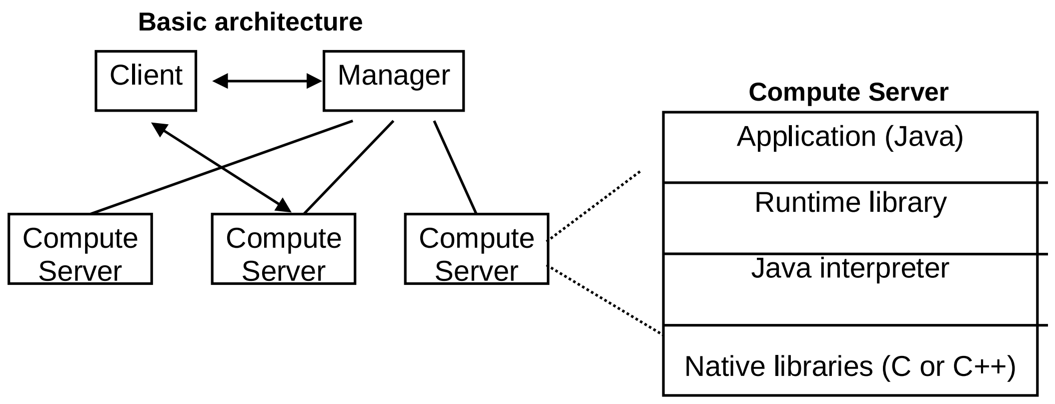

Figure 2.8: ATLAS system architecture (Adopted from [55])

The ATLAS system architecture consists of clients, managers, and compute servers as shown in Figure 2.8. A client with an application to run contacts the local manager to find an up (and idle) compute server. It then connects to this server to run the applications and provides feedback to the user while the application is running. During the execution, idle servers steal work from those that are busy, so that the application eventually spreads to all available resources. The runtime library in the compute server is responsible for work stealing, thread management, and marshalling and unmarshalling objects for communication to other servers.

The work stealing scheduler allows programs to run on a set of compute servers that grows and shrinks over time. The owner of a compute server can set the policy to determine when the compute server is idle. When idle, the compute server automatically joins the execution of a parallel program by work stealing. When the compute server is no longer available, all of its sub-computations are easily moved to another compute server. Moving a sub-computation requires only updating the information linking the sub-computation to its parent and child sub-computations.

2.4.2.4 Grid.It

Grid.It [56] is a project involving major research institutions in Italy aiming to providing innovative programming methodologies and tools. Within the Grid.It project, the ASSIST (A Software development System based upon Integrated Skeleton Technology) programming environment evolved from its very first version, only targeting workstation clusters, to the current version, targeting Grids and solving many critical problems related to expressive power, flexibility, interoperability and efficiency. ASSIST is a programming environment oriented to the development of parallel, distributed, high-performance, Grid-aware applications according to a unified approach. It provides the application programmer with a high-level language and its compiler, providing the need glue to effectively bridge different component technologies (e.g. CCM, Web Services). The compiler relies on an advanced run-time support dynamically enforcing Quality of Services constraints on the application by means of self-configuration and self-optimization features.

Grid.It and its ASSIST programming model present adaptation as a way to achieve a specified level of performance. An application should modify its structure or change the resources it has allocated when performance contracts are not satisfied. Within the ASSIST model, since adaptation is a way to achieve the contracted performance level, adaptation is mostly triggered from feedback control.

2.4.2.5 GridWay

The GridWay framework [57] has a number of similarities with the GrADS framework both in terms of concepts and the design of the architecture. The core of GridWay framework is a personal submission agent that automatically performs the steps involved in a job submission. GridWay job migration framework [58]takes into account the proximity of the execution hosts to the checkpoint and restart files. Their job migration framework also performs opportunistic migration and migration under performance degradation. Also, the user is required to alter their code in order to be linked to the Distributed Resource Management Application.

2.4.2.6 Grid Computing at the University of Leeds

The issue of adaptation was also addressed within the Leeds Grid research group in [59, 60]. In this work, the same approach to run-time prediction was used to support the job migration decision-making process. However, the run-time prediction method was initially developed within the context of the research presented here; see [15]. In [59, 60] the issue of supporting transparency, in the sense that the user need not alter their application code, was not addressed. Further, a description of the means by which the *application's* usage of resources can be monitored, rather than the resource itself, is not discussed. In contrast, the work presented in this thesis addresses both these issues, through the use of reflection to bind the adaptive service to the application code prior to job submission.

2.4.2.7 Proposed Adaptive Service and Related Projects: A Comparison

In contrast to the above research projects, the adaptive service in this research is situated between the system-level and the application-level. The user's application is bound to the adaptive service in the application level, using a reflective technique (see section 2.7), which provides adaptive capability to the user's application. Hence, the adaptive service runs at the same resource as the user's application. The adaptive service uses system's interface to collect information about the performance of user's application.

The approach proposed differs from the projects discussed above in the following ways:

- Decisions as to whether job migration is required are based on job performance rather than resource performance (e.g. node outages or resources being available). The decision as to whether to migrate or not relies on the output of the prediction model. The prediction model used here depends on the pattern of CPU time dedicated to an application on a specified resource.
- The adaptive service is bound to the application at job submission time, resulting in a reflective application, i.e. an application that can reason and make decisions about itself. Hence, adaptability is implemented in the broker in such a way as to isolate the user from the complexities of the

system. In particular, the user is not obliged to alter their code in order to achieve adaptability.

- The adaptive service runs in the same resource as the user job. Running the application and the adaptive service together on the same resource precludes the need for continual network communication as it has been proven in this research that adaptive service takes small fraction of the CPU load. If a decision is taken to migrate the job and this decision is made remotely then monitoring data must be sent to the decision-making component at regular intervals. This results in an increase in network traffic and also relies on the network performance to ensure that decisions are made on the basis of up-to-date information.

2.4.2.8 Data Replication

Replication has been studied extensively and different distributed replica management strategies have been proposed in the literature [61-63]. This is another approach to application performance optimisation. However, it complements rather than replaces the work presented here since it address different performance issues. Specifically, in the context of data Grid technology, replication is mostly used to reduce access latency, improve data locality, and/or increase robustness, scalability and performance for distributed applications. The number and placement of replicated data affects both the performance and the availability of a distributed database. The problem of optimising this aspect of the physical database design is known as replica management. Classic work on replica management concentrates on the file allocation problem [64, 65] - that is, the problem of finding optimal static data layouts. More recent work has been done to develop techniques that adjust the data layout dynamically, such as learning algorithms [66, 67].

2.5 Monitoring Application

Monitoring is the process of observing the behaviour of the system. When a user initiates the job, the job and the corresponding resource consumption needs to be monitored as it executes. There are two methods of monitoring the application during execution.

First, users can use the queuing utility to monitor the user's job, such as SGE. Most queues operate in the batch mode. In other words, when a user submits a job, the submission program immediately returns the user to the prompt and provides a ticket or job ID or token, which can be used to monitor the job at any later time. However, there are a number of disadvantages with respect to application and resource monitoring in queuing systems. For example, the monitoring requires the user to have an account on each machine, and knowing on which nodes the jobs are executing, only monitoring limited status of application and resource. Typically, in a Grid system, when a user submits a job, the job runs with permission of an ordinary user. Grid systems typically span multiple organisations and administrative domains. Often they run on machines that are controlled by queuing systems. Currently, there are no standard methods for monitoring the progress of jobs executed by Grid systems, and different systems use different techniques for monitoring applications and resources.

The second option is to use a monitoring utility like GMA [68], R-GMA [69], NWS [70], Netlogger [71].and Globus MDS[3] to name some. GMA is the Grid Monitor Architecture proposed by the Global Grid Forum (GGF)[72]. It consists of three components: consumers, producers and directory service. Relational GMA (R-GMA) is based on GMA from GGF. Its strength comes from relational model. Globus MDS aims to provide a standard mechanism for publishing and discovering resource status and configuration information. NetLogger is designed for analysis of communication inside applications. NWS provides accurate forecasts of dynamically changing performance characteristics from a distributed set of Grid computing resources. It can produce short-term performance forecast based on historical performance measurements.

Other monitoring systems are used for parallel application. Those system use structural approach SNMP [73] or hierarchical approach Ganglia [74]. Both approaches have three layers starting from distributed core which is complex with near cross-bar connections between nodes that provide high-level monitoring services to the user; to the middle layer which connects the local layer to the distributed layer; finally the local layer that gathers the data from locally available sensors.

The above systems solve the Grid monitoring problem to some extent. However, none of these systems provide any mechanism for monitoring the Grid applications execution without installing new software or changing the user code.

2.6 Migration

Checkpointing refers to the operation of saving the execution state of a running computation. In practice, checkpointing for applications is implemented either at system-level (e.g. MOSIX [75]), or is user-defined (e.g. Condor [76], CUMULVS [77], and MPI checkpointing [78], to name a few).

System-level checkpointing is a technique which provides automatic, transparent checkpointing of applications at the operating system or middleware level. The application is seen as a black-box, and the checkpointing mechanism has no knowledge about any of its characteristics. Typically, this involves capturing the complete process image of the application. The operating system must have the checkpointing feature. System-level is not suitable for the Grid environment because of resources with various operating system.. An example of system-level is MOSIX, which provides kernel-level checkpointing solutions. MOSIX is a set of kernel extensions which have been ported to Linux. MOSIX uses a kernel module to provide transparent load balancing and process migration.

User-defined checkpointing is a technique that relies on programmer support for capturing the application state. The approach is not transparent to the user, if the user has to alter code, but is more flexible for the same reason. A more detailed, comparison between the two approaches can be found in [79]. In summary, although system-level checkpointing is transparent to the user and easy to use, it is less portable and flexible, and creates larger checkpoints as compared to user-defined checkpointing. In the past, most of the checkpoint schemes were supposed to be transparent to the application, and implemented at the system level. More recently, user-level schemes have been explored in greater detail.

In Condor-G, if and when any policies are violated during the execution of a job, it has the ability to migrate a job to another location. It does so by checkpointing the state of the process, and then restarting the process on another machine with the same image. The checkpointing is implemented at the user level, with no modifications to the kernel code. The biggest advantage of using Condor for checkpointing and migration is that it is transparent to the user. The user only has to re-link his or her code with the Condor checkpointing libraries. This works fine for users who have access to the software, but can be a hindrance to users of third party software. However, there are several disadvantages with Condor's approach to checkpointing and migration. The process image for a job may be huge, and a lot of unnecessary information could be stored if checkpointing is done at the process level. Process images are also highly dependent on the architectures, and hence can not be migrated across heterogeneous platforms. It also does not handle migration of a set of communicating processes (using signals, sockets, pipes, _les, or any other means). Despite these shortcomings, a wide variety of real-world user codes can be accommodated by this approach.

CUMULVS provides a user-level mechanism that assists in creation of checkpoints, and restart from saved checkpoints. The user application selects the minimal program state necessary to restart or migrate an application task. Application task’s can then be migrated across heterogeneous architectures to achieve load-balancing or to improve a task's locality with a required resource. It is suitable for several scientific applications which need checkpointing only at a coarse-grained level. Also, CUMULVS does not do anything special in order to handle distributed applications. Instead, it relies on the application writer to create a globally consistent checkpoint. This increases the expertise required on the part of the application writer.

As this research is concentrated on adaptive service technique, the migration aspects is used as the way to save the state of the job without trying to develop the state of the art. This research looked at other projects involved in the migration, and implements an approach that would prevent the users from altering their code. The migration is not easy to achieve in the Grid, as the resources are heterogeneously distributed under different administration with different policies.

2.7 Reflective Technique

Smith introduced the concept of reflection and that of a computational process that can reason about itself and manipulate representations of its own internal structure [80]. Two properties characterise reflective systems: introspection and causal connection. Introspection enables a computational process to have access to its own internal structures. Causal connection enables the computational process to modify its behaviour directly by modifying its internal data structures, i.e. there is a cause-and-effect relationship between changing the values of the data structures and the behaviour of the process. The internal data structures are said to reside at the metalevel, while the computation itself resides at the baselevel; thus the metalevel controls the behaviour of the baselevel.

Reflection provides a principled means of achieving open engineering, i.e., of extending the functionality of a system in a disciplined manner [16]. A key attribute of reflective systems is that of separation of concerns between the metalevel and the baselevel. For example, [20] incorporated replication techniques into objects using the reflective programming language Open-C++. The implementation of the replication techniques was performed at the metalevel, with few changes to the underlying baselevel application. The design and implementation of the replication techniques were separated from the design and implementation of the actual application, thus allowing the replication techniques to be composable with many applications. In general, reflective architectures enable the composition of non-functional concerns with the underlying computational process [81].

Another advantage of reflective architectures is that they enable flexibility and extensibility of functionality. Reflective architectures have been used in such diverse areas as programming languages [82],[83], [84], [85], [86],[87], operating systems [88], real-time systems [89], [90],[91], fault-tolerant real-time systems [92], agent-based systems [93], dependable systems [94], and distributed middleware systems, e.g., OpenORB [16], FlexiNet [95], OpenCorba [96] and Legion [97].

A feature common to all reflective systems is that they answer two questions: What internal structure or metalevel information (meta-information) is exposed to developers? How does one access the metalevel? The answer to the first question is application-dependent. For example, in real-time systems such as Fault tolerant Entities in Real-Time (FERT) or Spring [92], [90], the meta-information includes timing constraints of tasks, deadlines, and precedence constraints. In a programming language such as Common Lisp Object System (CLOS), the meta-information includes slots and methods [84]. In an object-based distributed system, meta-information can include methods, arguments and replies [16], [95], [96]. The answer to the second question also varies. A popular method of programming the metalevel is through an object-oriented paradigm in which a metalevel object defines and controls the behaviour of baselevel objects [82], [84]. Other means of accessing meta-information include using compiler technology [20], [98],[83] configuration files [86],[87], and events [97], [99].

2.8 Summary

The Grid has definitely been conceived and born. How far along in its development is not clear. However, the potential is clear, in the short time the Grid has been conceived, interest has grown considerably. This is due to what the Grid can provide for scientists of many disciplines.

This chapter describes the Grid and its existing technologies with their application. Also, the work been done in identifying the issues involved with adaptivity in the Grid. Moreover, the chapter has established the infrastructural requirements for the development and deployment of adaptive service. The adaptivity in the Grid is the key issue of this research, and is a little developed field.

Next is the design of the proposed framework. This includes the motivating scenario, Grid resource broker components, toward adaptivity, adaptive service components, and finally the role of reflection technique in the framework.

Make everything as simple as possible, but not simpler.
— Albert Einstein (1879-1955)

Chapter 3

Design of Adaptive Grid Resource Brokering

3.1 Introduction

Today, as Grid computing is becoming a reality, there is a need for managing and monitoring the available resources, as well as the need for adapting the user's job to the changes in resources' performance. Figure 3.1 shows the highly abstract picture of the architecture proposed to show the flow of the user's job with the adaptive service bound to it.

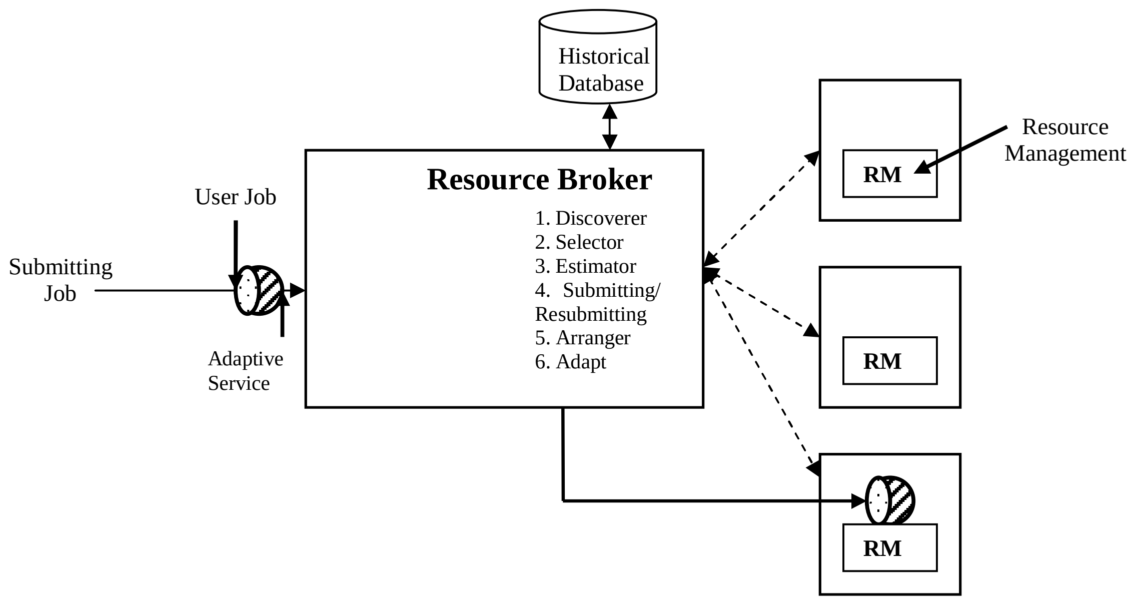

Figure 3.1: Adaptivity in Resource Broker

The proposed architecture supports run-time adaptation through the use of an adaptive service. As indicated in Chapter 1, adaptation is important to address the dynamic and heterogeneous nature of the Grid. This is a key issue both before an application begins execution (adaptive resource selection) and during run-time (run-time adaptation). It is envisaged that Grid implementations will operate within an economic framework [100]. This means cost/performance trade-off decisions must be made at run-time, requiring mechanisms to support performance prediction.

The adaptive service supports job migration during run-time to ensure timely job completion. Performance prediction is used to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user-specified deadline.

This chapter discusses the design of the resource broker and adaptive service, as well as how to integrate the adaptive services into the resource broker. The resource broker runs on top of the Globus Toolkit. Therefore, it provides security and current information about the available resources and serves as a link to diverse systems available in the Grid.

The chapter starts with a motivating scenario, then describes the design of the resource broker and adaptive service. Then follows the role of reflective technique in binding the user's job with the adaptive service.

3.2 Job Requirements

Certain jobs need a particular environment during execution. This environment needs to be set-up before the job actually starts executing on the remote node. For this purpose, the requirements element is provided. It can be used to specify a set of initialisation tasks.

Requirements are of two types: node and job. A node requirement is a set of tasks/conditions that need to be satisfied before a node can be used for submission of jobs, such as the hardware platform and operating system. So, a node requirement is performed by the broker before any jobs are submitted to it. This is done once and only once for each node. A job requirement is also a set of tasks/conditions which are to be performed once for each job, such as job execution time, number of CPU with preference of type of speed, minimum RAM and disk space. Other job requirements include the execution start time and its duration, and the location of both the executables and any data the job depends on. This stage is pivotal, as it will influence the following stages, determining where the job will eventually be submitted for execution.

3.3 Job Submission

There are many types of Grid jobs that the resource broker handles e.g. interactive, MPI (Message Passing Interface), and batch, to name a few. However, this research focuses on batch jobs, which is the type handled in the DAME project, as explained in Section 1.4.

A batch job can be submitted directly to the local scheduler or/and use the broker through SGE RSL (Resource Specification Language) [44] script, which is a language for expressing user resource requests (e.g. CPU number, maximum execution time, and maximum wall time), as well as configuration for the application (e.g. how to execute the job). The script usually consists of the run-time duration, the directory location where the job's executable files are stored, and the location of any data the job depends on.

Submitting the job directly to the local scheduler can be done using two methods. Firstly, the user can submit the job to the selected queue for execution. This requires the user to know which queue can handle the job. The user can query the available queues, determine their loads, and decide which queue is suitable for their job. Alternatively, the job can be submitted to SGE without specifying the queue. SGE allocates the resources based on the job description. The disadvantage of these methods is that the user needs to log into the machine and to be knowledgeable about the SGE command. Also, the user has only the choice of the machine he/she logged on to.

3.4 Motivating Scenario

In reference to the motivating scenario in Section 1.2.2, the DAME maintenance team does not want to be involved in the complexity of changing the source code or even if the team were willing to change it, then they may have no control over it. In order to bind the user's job with the adaptive service, the reflective technique is used. Hence, the user does not need to alter the code.

However, suppose the DAME maintenance team wants to submit a job which needs to finish in 1 hour with 90% CPU power. Having received a user's request to run the job and details of its requirements, a resource broker queries various sites on the Grid that encompass resources which could cater for the user's request. The broker then transfers the data and submits the job for execution. However, while the job is running monitoring is important in case the performance of the resource is degraded, e.g., another job starts to run the same resource. If the performance is degraded, the job will not finish within the time specified and needs to be migrated to a better resource.

The adaptive service has the mechanism to ensure the job will finish at the time specified, and reflective technique applies the binding without the need to change the source code.

3.5 Adaptive Grid Resource Broker Infrastructure

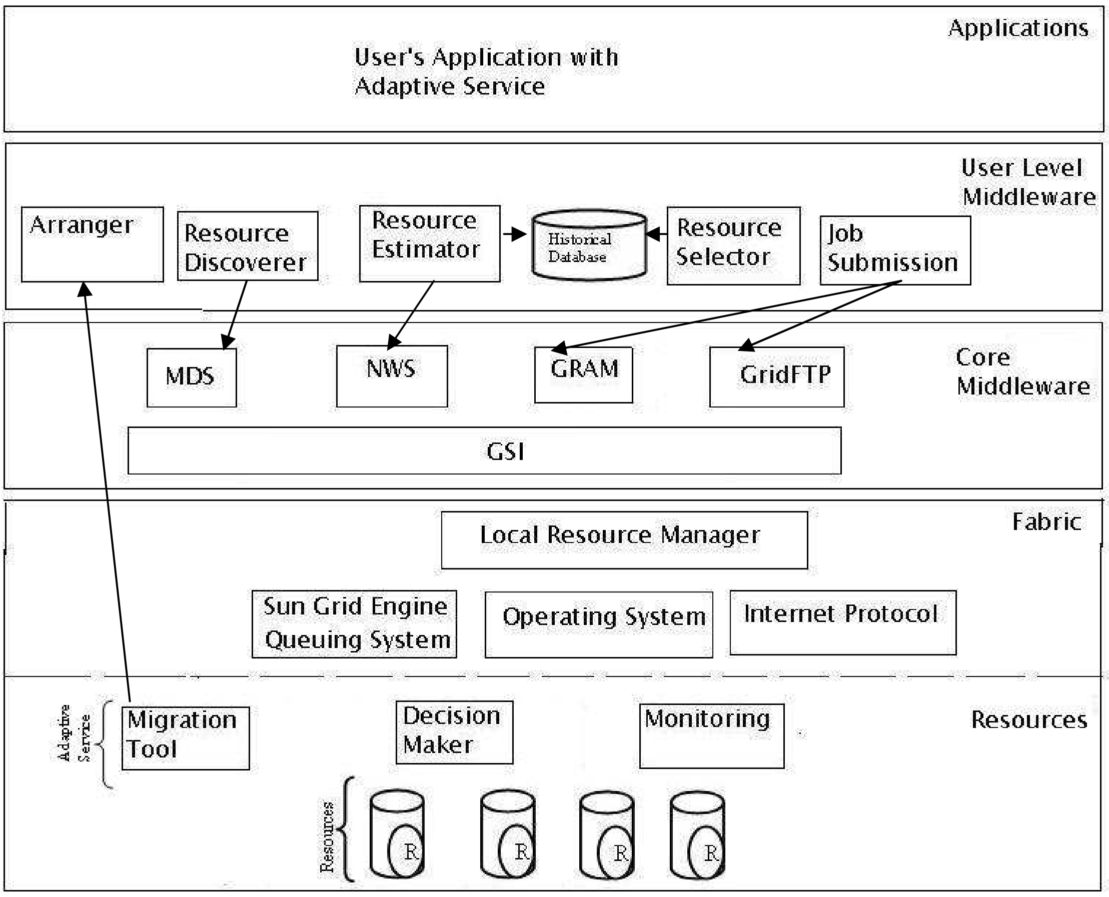

Figure 3.2: Adaptive Resource Broker and Globus Toolkit Components

Figure 3.2 depicts the high level picture of the infrastructure of the Adaptive Resource Broker proposed in this research. The infrastructure consists of the following:

- The application level, where the user's application is bound to the adaptive service using reflective technique at submission time, resulting in a reflective application, i.e. an application that can reason and make decisions about itself. Hence, adaptability is implemented in the broker in such a way as to isolate the user from the complexities of the system. In particular, the user is not obliged to alter their code in order to achieve adaptability (see Section 3.9).
- User level middleware, where the resource broker components reside. The components include resource discoverer, selector, estimator, arranger, and job submission. The resource broker is the middleware between the reflective application and the Globus toolkit (see Section 3.6).
- Core Middleware, where Globus toolkit is used as Grid middleware in this research. The components used are shown in core middleware level including MDS which is used by resource discoverer to enquire about the available resource, GridFTP for data transfer and GRAM for submitting the job to local scheduler. The NWS is used to estimate the transfer time of the job and the related data.
- Fabric, which consists of all the globally distributed resources that are accessible for this research. These resources are computers (such as clusters, or parallel computers) running a version of UNIX as operating system, storage devices and Sun Grid Engine as the queuing system. The adaptive service is running in the resource; it consists of the monitoring tool, migration engine, and decision manager. Notice that the only component that communicates with the broker is the migration tool in case of requesting better resources.

3.6 Grid Resource Broker

The resource broker is developed to simplify the job submission, resource discovery and selection for users, in order to insulate the user from the Grid middleware complexity.

The resource broker developed for this research (depicted in Figure 3.2) differs from traditional batch system schedulers/brokers [4, 22, 23, 101] etc, in that every user has his/her own copy of the broker to have the ability to handle user's job individually. The aim of such a personal broker is to find, characterise, select, and (co)allocate the resources best suited for each job submitted by the user. This section consists of a general overview of the resource broker, description of job requirements, then the main components in the resource broker.

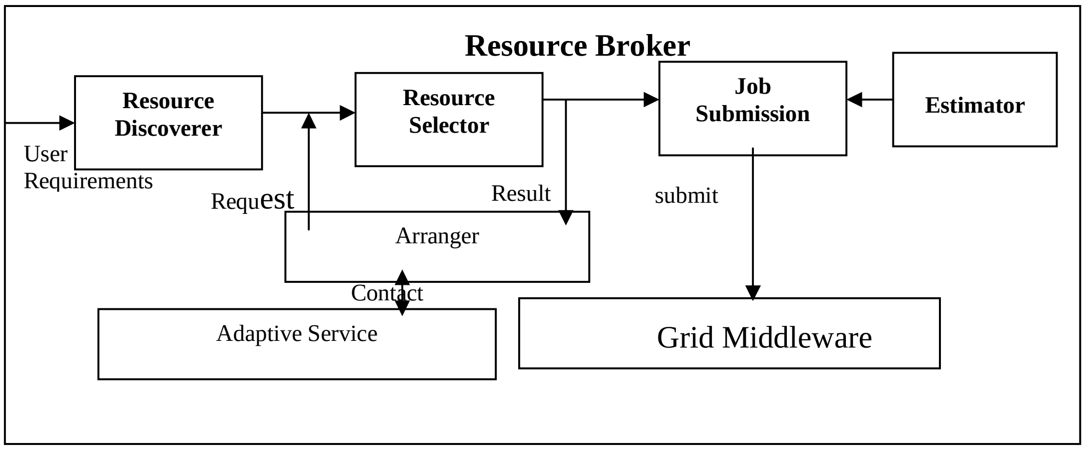

Figure 3.3: Proposed Resource Broker Architecture

3.6.1 Overview

A general purpose resource broker that facilitates the user's resource selection and job submission automatically is required. The resource broker uses Globus toolkit (GT) as a Grid middleware. While GT provides effective means to aggregate and virtualise Grid resources, the discovery and categorisation of vast resources in this heterogeneous and dynamic environment presents a problem for the end user, due to the complexity of the information involved. Also, the Grid Information Service [37] provides an overview of the available Grid resources as well as information about the current status of the Grid, an average user may be overwhelmed with the information to process, or the user may not have enough experience to select the best available resource. Automation of this selection process would simplify and expedite this process.

3.6.2 Historical Database

The aim of the historical database is to study the performance behaviour of the Grid resources. The data comes from all previously run jobs. Such data include number of jobs failed, jobs submitted and total queuing time. The database is stored in MYSQL[102]. Further detail can be found in chapter 4.

3.6.3 Resource Discoverer

The Resource discoverer is the component that contacts resources that may be suitable for the user’s application. The resources that do not meet the minimum requirements are eliminated. This process enhances efficiency by not considering resources that do not have the ability to handle the job’s requirements.

This stage queries the current state of those resources that have been identified as candidates. This can be done by contacting the information providers associated with those resources that the user has the credentials to use and are able to meet the job’s minimum requirements. The information provider that has been used for this research is MDS, mentioned in Section 2.3.1.4.

```
Mds-Cpu-Free-15minX100
            100
      Mds-Net-Total-count
            2
      Wrg-Sge-Queue-name
            all.q@testgrid6.leeds.ac.uk
      Wrg-Sge-Queue-loadavg
            0.01
      Wrg-Sge-Queue-type
            BIP
      Wrg-Sge-Queue-slots
            0/1
      Mds-Memory-Ram-Total-sizeMB
            241
      Mds-Net-Total-count
            2
      Mds-Cpu-Free-15minX100
            100
      Mds-Memory-Ram-Total-sizeMB
            241
```

Table 3.1: Data Reported Back from the Information Providers

MDS provides both static and limited dynamic information. The resource discoverer interprets and evaluates the data returned from information providers, which is in the form of a list of attributes with corresponding values (Table 3.1). The selected resource passes to the next stage for more evaluation.

3.6.4 Resource Selector

The resource selector makes use of a historical database which contains data extracted from previous runs, data such as number of jobs, number of jobs failed, and total queuing time (more on the historical database in the next chapter). After receiving the list of the available resource from the previous component, the resource selector searches the database for historical information about the candidate resources, and the resource with higher probability of failure will be eliminated.

The aim of the resource selector is to minimise the likelihood that the resource fails to meet the user's requirements. However, a further option guarantees the requirement has been met during the run-time using the adaptive service.

3.6.5 Estimator

The Estimator component (or simply the estimators) is used to predict the resource consumption of a job. The Estimator provides the following estimators:

3.6.5.1 Queuing Time Estimator

The Queue Time Estimator estimates the time a task will spend in a queue waiting for its turn to start execution. This queue time is used by the adaptive service to know how much time the job will spend in queue in case of migration. Moreover, the queuing time is added to the elapsed time which is used by the decision manager (more detail in the next chapter).

In order to estimate the queue times the following sequence of events take place:

1. The resource selector passes the name of the resource that has been selected to the estimator.
2. The estimator then retrieves from the historical database the mean total queuing time.

3.6.5.2 File Transfer Estimator

Since transferring files could be a time consuming operation, there is a need for an estimator method that will tell the adaptive service how much time this file will take to transfer. For transfer time estimation, first, determine the bandwidth and latency between the resources, the resource broker uses Network Weather Service (NWS), and then calculates the transfer time using this bandwidth and the file size (more in the next chapter).

3.6.6 Arranger

This component is part of the resource broker and is responsible for arranging all interaction between the resource broker and the adaptive service. It is the most important component toward the integrity between resource broker and adaptive service. There are two procedures the arranger handles with resource selection.

- In the event that migration is required, the arranger component communicates with the resource selector. Note that the resources that have the ability to handle the user's job have already been identified by the resource selector.
- The arranger then instructs the broker to resubmit the job to the resource selected. This includes transferring all the files from the old to the new resource.

In the next section, the design of the adaptive service is introduced. This includes the design of the tool and process that has been developed to ensure the user's job finishes within the time specified.

3.7 Towards Jobs Adaptation

The common point about adaptation is that it consists of changing some things in applications. This asks three major questions: why adaptation should be done, where it is done, when it should be done.

3.7.1 Purpose of the adaptation

The goal of adaptation in this research is to optimise the performance of the user's applications in order to meet the requirements. The adaptation is to evaluate the resources and, if necessary, allocate the user's application to a better resource. Furthermore, the adaptive service considers the overhead in both the resource and network. This is very important as the adaptive service might be a burden to achieving the user's requirements. Unlike other projects, the user does not have to change code to achieve the adaptation.

3.7.2 Location of Adaptation

The adaptation dispatches with the user's job to the selected resource and adapts when the resource performance changes. Hence, the adaptive service is bound to the user's job and both execute in the same resource. The location of the adaptive service in this research is useful to reduce the network traffic and be more accurate in terms of timing as the traffic delay might effect the prediction model results in case the adaptive services run remotely. Hence, the adaptive service developed for time constraint jobs.

3.7.3 Decision to Adapt

Decision to adapt is different for each project mentioned in Section 2.4.2, but the common decision is to migrate. The proposed adaptive framework includes our own monitoring tool. This tool measures the resource performance and feeds it to the decision making. The information from the resource includes how much the job consumes from the resource as our adaptive service applies to time constraint jobs. The difference from the other projects is that our decision based on the prediction module developed for this research.

3.8 Adaptive Service

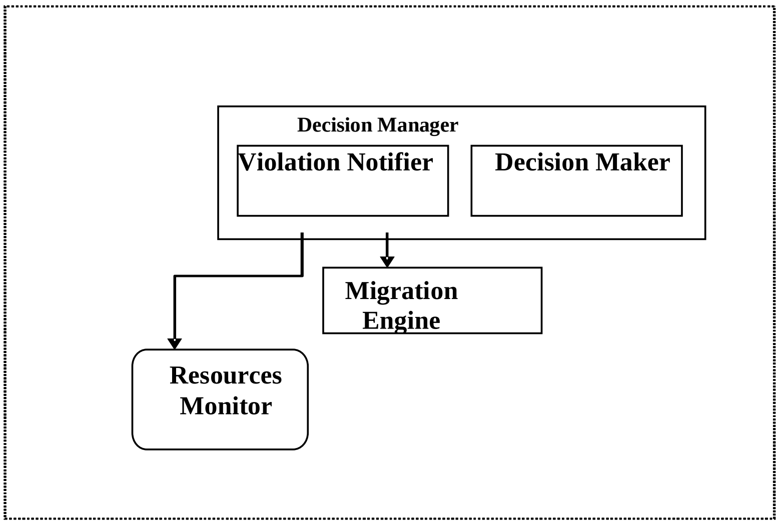

Figure 3.4: Adaptive Service Architecture

The adaptive service (depicted in Figure 3.3) fits into an overall framework that aims to address problems that arise in meeting particular requirements in a Grid computing environment. Specifically, a scenario is envisaged where

- The user application has a time constraint.
- The user requires flexible management functionality.

The adaptive service is designed based on scenarios in Section 1.2.2 and 3.2. The adaptive service is a key feature of the architecture presented in Chapter 4, since without it fulfilment of the user requirements depends entirely on the dynamically varying performance of the resources. A number of issues need to be considered in deciding on an appropriate design of this service.

Firstly, support for resource selection is required in order to make use of the adaptive service. This is necessary both before run-time and during execution if migration is required. The broker and adaptive service communicate through an open port, e.g. when migration of an application to an alternative resource is required.

The adaptive service needs to be implemented in such a way as to account for and minimise overheads. Firstly, the adaptive service itself must be implemented in such a way as to ensure that its own presence does not have a negative impact on the application performance (e.g. by significantly increasing the execution time). In addition, overheads associated with job migration (e.g. file transfer times) need to be accounted for when deciding whether migration is an appropriate course of action. Further, in order to make use of the adaptive service the user should not be required to alter their code prior to submission; the system complexities should be transparent to the user.

The adaptive service includes components for supporting job migration to an alternative resource. Decision-making algorithms are used to determine, based on monitoring information, whether an application is predicted to meet the user's requirements. For example, if the application CPU usage declines during run-time, a decision may be made that migration is required to ensure completion within a time specified. In this case, an adaptation request is made by the adaptive service to the resource broker. If alternative resources can be found, the broker informs the adaptive service, the job is then saved (using checkpointing) and the adaptive service instructs the broker to migrate to an alternative resource.

3.8.1 Monitoring Tool

The first attempts at designing the monitoring tool are focused on identifying existing open-source monitoring tools, which are minimally modified in order to be integrated within our monitoring tool. This approach allows us to focus on the novel aspects of our architecture, instead of dealing with already solved issues. The novel aspect is to locally monitor the application with very up-to-date data, very minimal overhead and real-time monitoring (no storage needed to store the past data).

The main characteristics of the monitoring tool are:

- Application monitor: It is important to note that the monitor collects data about the application performance rather than general resource information. For example, the Network Weather Service (NWS) provides information about the overall CPU usage, whereas the decision maker requires the CPU usage of the user's application.

- Locality: The monitor tool resides in the resource where the user's job is executed. This way, the overhead in the network is reduced, which speeds up the process of collecting and filtering resource data.
- Accuracy: The resource data collected are accurate and up-to-date. The data are collected from the local operating system, hence not using third party software.
- Real-time: The information collected by the monitoring tool is specifically used for the adaptive service. The monitor is not designed as a general purpose tool for use by other programs. Hence, the monitor starts working when the user's job starts to execute, and terminates when the job is finished or migrates to another resource. The data collected by the monitor tool is not stored in any storage but feeds the decision manager.

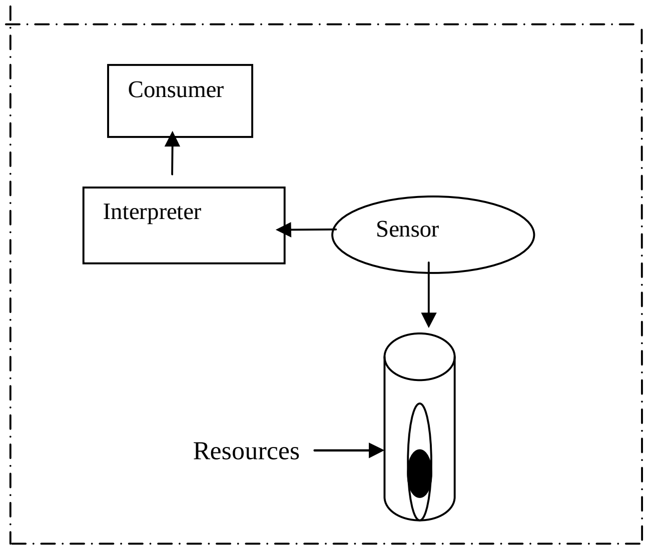

Figure 3.5: Monitoring Architecture

The monitoring architecture consists of the components shown in Figure 3.4 and they are as follows:

3.8.1.1 Sensor

A sensor is a program that generates a time-stamped, performance-monitoring event. The sensor performs monitoring tasks at regular intervals, e.g. CPU load. Hence, the sensor resides in the resource as the user's jobs executed. Moreover, it uses the local operating system to gather the information about the state of the user's application.

3.8.1.2 Interpreter

Since the monitor performs in real-time, the data collected by the sensor has to be interpreted for the consumer (decision manager). The interpreter needs to calculate the current and mean CPU usage from the raw data to pass it to the decision manager.

3.8.2 Decision Manager

The decision manager (Figure 3.3) is responsible for both decision-maker, to determine if a job migration is required, and notifier, which informs the resource broker of actual violations of the user requirements and the need for a better resource. Hence, it provides an interface to communicate with external components. The information required to make these decisions is obtained from the monitoring data, and information obtained from the historical database. The monitoring tool provides data at time intervals, data such as main CPU usage and utilisation.

In order to make use of the adaptive service, it is necessary to have a decision-maker algorithm that initiates a search for alternative resources when the performance degrades to an unacceptable level. The algorithm provides a prediction time of how long the job takes to finish with the available resource usage. The decision algorithm considers the following:

Batch queue waiting. Once all the files are located on the resource, the application can start to execute. As the resources are operated by a local scheduler, the job may have to wait for other jobs to complete before it can start. The batch queuing time in the new resource after migration will be considered. The delay is referred as batch queue waiting time.

Program execution. The time required to execute the application on the resource varies with the characteristics of the job, the performance of the resource, and with how apt the resource is for executing the job. Note that both the time taken (in the event that migration takes place) on the initial resource, and the time taken on the new resource, contributes to the overall execution time. For efficient usage of the available Grid resources, the time required for program execution should be large compared to the other parts of the total time to finish the job.

Files transfer. The time it takes to transfer the file to the selected resource is important toward time-constrained jobs. The RSL job description specifies where each file should be stored. The file transfer time is part of user's application execution time. In case of migration to a new resource, the files transfer time is considered in the decision maker. This research uses NWS to estimate the time to transfer all the files to the resource.

The decision manager bases the decision on the result from the prediction model, queuing and transferring time, and the input data from the user. If there is a need to migrate, then the decision manager makes sure there is another resource better than the current resource, and then notifies the migration tool.

3.8.3 Migration Engine

The migration tool ensures the state of the job is saved and the job is terminated. Migration is performed typically for relocating a user's job to a better resource for improving the performance of the application or resource owners. The design of the migration tool involves the following:

- **Portability:** Grid resources are heterogeneous by definition, since they span various administrative domains. Hence, no assumptions can be made about their architectures. This implies that the job's state has to be stored in a platform and architecture-independent manner, so that it remains portable across the Grid.
- **Checkpoint Size:** It is desirable that the checkpoint size is minimal, so that the checkpointing and restart, and migration implementations are efficient.
- **Scalability:** The architecture should scale well with the number of components that are part of an application, and also with the number of applications themselves.

The design of the migration tool consists of migration process and checkpointing process. Those two processes form a migration tool and are describes as follows:

1.***Migration Process:*** this process is defined as terminating the user’s job and arranges a message to be sent to the broker for the purpose of relocating all the files to the new resource. Also, this process transfers the files produced by the checkpointing process to the resource broker.

2. ***Checkpointing:*** checkpointing is defined as the process of saving the state of the user’s job. Hence, the user’s job does not need to start from the beginning again in the new resource. Checkpoints are created on the remote nodes, but stored on the resource broker node. Thus, if a job finishes prematurely on a remote node, e.g., the remote node crashed or the load became too high causing the job to be terminated prematurely, the resource broker can resume the program from the most recent checkpoint on a different resource.

The first decision in designing checkpoints into a Grid system is to consider how the checkpoints will be created. There are three main options to consider:

- Allow the system to arbitrarily create a checkpoint when it needs to.
- Allow the user to specify when a checkpoint should be made.
- Both options 1 and 2.

Option 1 would appear to be the best, and is the one used in the Condor System (see Section 2.4.1.1). This option has the advantage that the user does not have to decide when to create a checkpoint for his/her application. This option can decrease execution time if the checkpoints are expensive to make and save for future use. The downside of this option is that the system must have the ability to checkpoint.

Option 2, allowing users to create their own checkpoints, is more feasible. In the end, only the programmer knows when her/his application is in a safe state to create a checkpoint. The programmer is also in the best position to determine if there is any real use in creating a checkpoint for the application. For example, if the application is only going to take six minutes to execute, there may not be any benefit in using checkpoints, since it would not take too long to just restart the application from the beginning. On the other hand, if the application is going to run for six weeks, checkpoints could become invaluable. The downside of having the user indicate when to create checkpoints is that the user needs to customise the application code to work within the adaptive service. Hence, the user must know how the adaptive service works.

As indicated in the previous chapter, the migration engine does not use any ready tools and do not change the user's application to enable migration and checkpointing. The adaptive service is responsible for the migration and checkpointing by saving the state of user's job, then transferring it to the new resource, and finally, restarting the job from where it stopped. This method is similar to system-level in such a way that the adaptive service saves the state of the job in a regular interval.

Migration typically does not result in any loss of computation. In other words, the job need not be rolled back to a globally consistent state from the past. Only the last stored state of the job will be migrated.

3.9 Role of reflection technique

The adaptive service implementation involves binding the adaptive service to the application before run-time, using a reflective technique. The user is not required to alter their code. The adaptive software is bound to the application at job submission time, resulting in a reflective application, i.e. an application that can reason and make decisions about itself. Hence, adaptability is implemented in the broker in such a way as to isolate the user from the complexities of the system. In particular, the user is not obliged to alter their code in order to achieve adaptability.

The essence of the reflection paradigm is a system that it is self-optimising, modifying its behaviour. The reflective paradigm is introduced into object-oriented programming using the meta-objects protocols [82], where the functional and non-functional aspects are separated, using user's job (base-object) and adaptive service (meta-object), respectively. A base-object describes the application functionality, while the associated meta-object executes the control policies that determine the behaviour of its corresponding base-object. Figure 3.5 depicts the binding of the application code to the adaptive software prior to run-time.

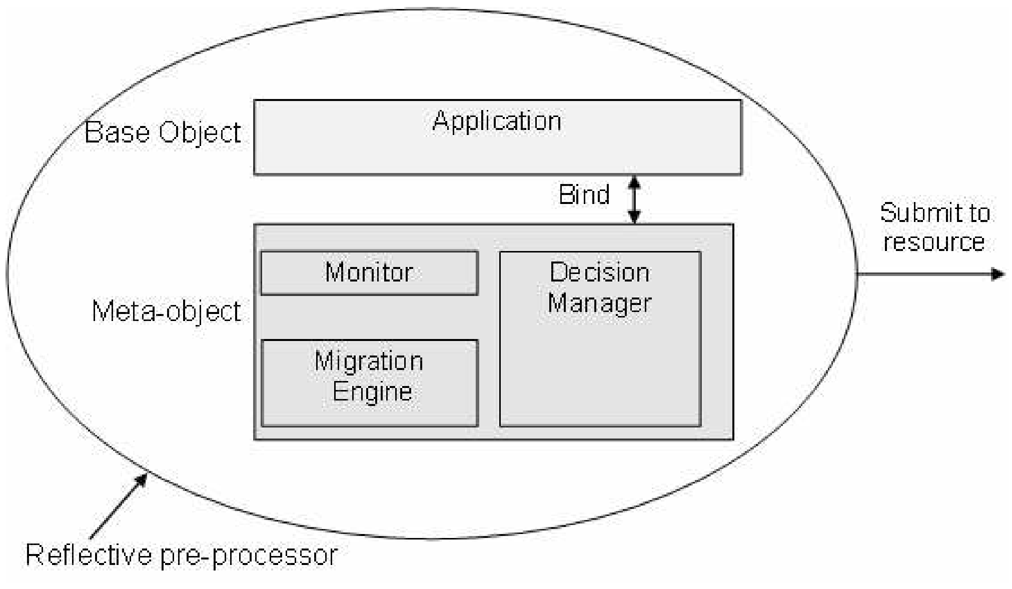

Figure 3.6: Binding of adaptive components to application through reflection

The binding is carried out by a reflective pre-processor, which does not require any knowledge of the user's application. The result is a reflective application which is capable of reasoning and making decisions about itself. Specifically, the meta-object enables decisions to be made during run-time about the base (application) object. In this case, if the decision manager determines that the application should be migrated, then the migration engine checkpoints the job and a request is sent to the resource broker for an alternative resource.

One clear advantage of this approach is that it reduces the required network communication, since the monitoring and decision-making tools reside in the resource(s) being used by the application.

3.10 Summary

This chapter describes the design of the architecture depicted in Figure 3.1. Our framework consists of two parts: the resource broker and the adaptive service. The purpose of the resource broker is to hide the Grid middleware complexity, and to provide the resource discovery and selection. The resource discovery chooses the resource that is able to handle the job's requirements; such requirements as memory, operating system, and CPU. The resource discovery contacts the resources by querying their state from the information service. These resources are checked against the resources in the historical database to study the behaviour of the identified resources in the past.

The second part is the adaptive service, which includes monitoring, decision making, and migration tool. The tools were designed with consideration of purpose, location, and means of adaptation (Section 2.3.3). A monitoring tool was designed to monitor the jobs' usage, but not the overall resource behaviour. The decision manager ensures there are better resources before deciding to migrate. The main purpose of the adaptive service is to ensure the job finishes within the time specified.

Running the application and the adaptive service together on the same resource precludes the need for continual network communication. If a decision is taken to migrate the job, and it is made remotely, then monitoring data must be sent to the decision-making component at regular intervals. This results in an increase in network traffic, and also relies on the network performance to ensure that decisions are made on the basis of up-to-date information. The approach presented here avoids this potential problem with any overheads associated with the adaptive service monitoring and decision manager, which slow down the application, since they are sharing the CPU. This does not appear to be a serious issue, as discussed in Chapter 5.

The best way to predict the future is to invent it.
— Alan Kay

Chapter 4

Implementation of Adaptive Service

.

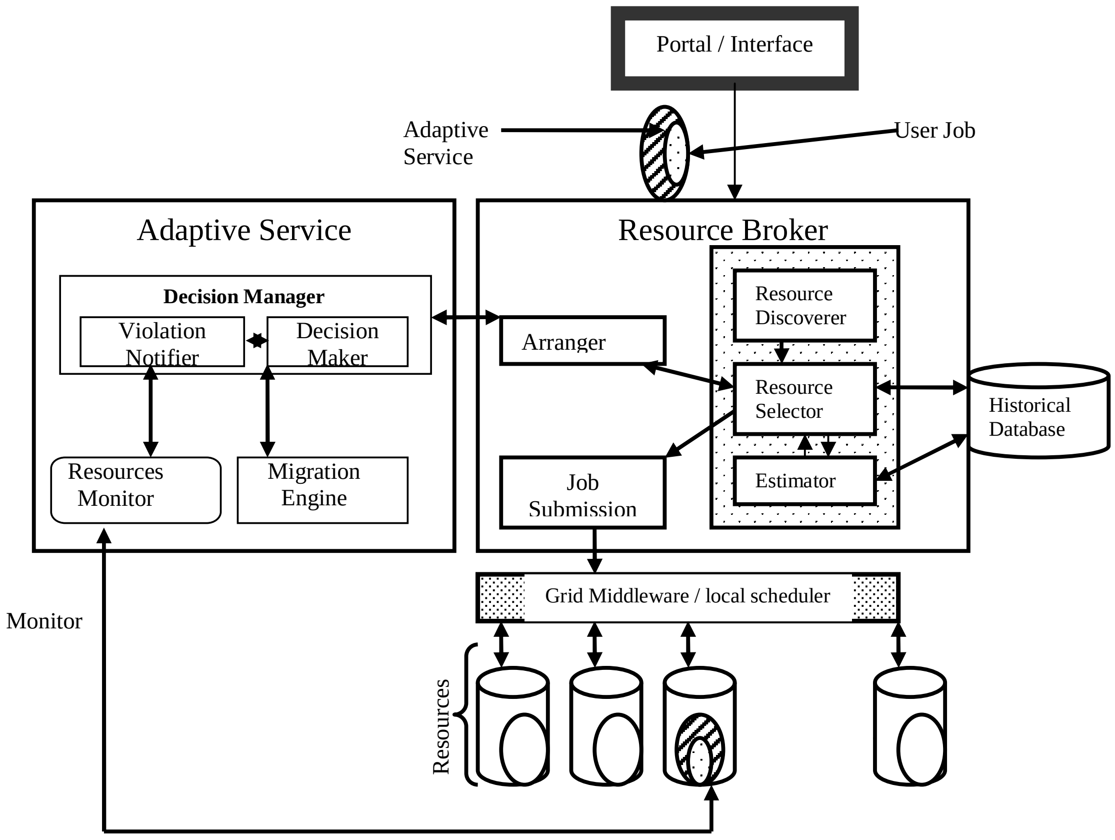

Figure 4.1: Adaptive service with resource broker integration.

4.1 Introduction

The previous chapter described the design of the adaptive resource broker. The resource broker and the adaptive service are integrated to ensure the user requirements are met. Figure 4.1 depicts the integration of the adaptive service and the resource broker. The architecture shows the components that comprise the adaptive service, the resource broker, and the binding of the job with the adaptive service

This chapter starts with discussion on the programming language used for implementing the adaptive service then the reflective technique, followed by the use of the historical database toward the adaptation. The monitoring, prediction model, decision manager, and migration are then discussed in detail as the adaptive service's components.

4.2 Programming Language

This section gives a brief review of the Java programming language used for this research. Java [103] is simple, object-oriented, interpreted, dynamic and robust. The Java virtual machine is simple and general enough so that it can be implemented on most modern processors. A Java application in bytecode can run on any system supporting the Java Virtual Machine (JVM). Java bytecode is therefore portable and conforms to the "write once, run anywhere" philosophy. This feature is of particular significance when considering the heterogeneous nature of the Grid. Note that the adaptive service is written in Java, but the user's job could be in any other language, such as C.

It is important to note that JVM is an abstract machine which provides a set of basic operations based on a simple, stack-based operation model. JVM is implemented on a wide range of processors ranging from those used in embedded systems to those used in supercomputers.

4.3 Binding Procedure

As mentioned in section 3.5, the adaptive service is bound to the user's job prior to run-time: this research uses Javassist [18] as a toolkit to produce the reflective system. The main advantage of using javassist is that the user's job and adaptive service do not need to be altered in order to bind them together. Javassist has the ability to inspect, edit, and create Java binary classes. The inspection aspect mainly duplicates what is available directly in Java through the Reflection API but having an alternative way to access this information is useful when modifying classes rather than just executing them. This is because the JVM design does not provide any access to the raw class data after it has been loaded into the JVM.

Javassist provides object-oriented frameworks for writing programs that manipulate the structure of Java class files. They provide load-time representations of elements of class files such as methods, types, instructions, etc. Java programs can then be written that describe how class files can be rewritten as late as load time.

```
…
…….
public class Main {
   public static void main(String[] args) throws
Throwable {
       Loader cl =
(Loader)Main.class.getClassLoader();
       cl.makeReflective("UserJob",
         " ClassMetaobject");
       cl.run("UserJob", args);
   }
…………..
```

Figure 4.2: Binding Specification

The binding itself is described by a binding specification (Figure 4.2). This registers UserJob as a reflective class. The class loader modifies the UserJob.class when loading into the JVM so that the outcome is a UserJob object controlled by an AdaptiveService object.

The next section describes the role of the resource broker in achieving the adaptation within the Grid environment. The importance of the resource broker in adaptation comes from preparing the job to run on a reliable resource. The historical database provides ability for the resource broker to select a resource that would handle the user’s job. Those features distinguish the resource broker developed for this research from other resource brokers mentioned in Section 2.3.

4.4 Historical Database

The aim of the historical database is to provide information about the past performance and the behaviour of resources. The historical database is used to enhance the resource selection process, and thereby increase the likelihood of timely job completion. Moreover, the database supports the queuing time prediction. It is important to note that it is assumed a user only has access to data relating to their jobs that have previously run (i.e. not other users jobs).

The historical database is stored in a MySQL database [102]. This information is used to estimate the likelihood any given resource will fail to meet the user's requirements during execution time. For example, if a resource often fails, it is unlikely to be selected, compared to a resource that never fails. The resource selector makes use of the historical database by estimating the probability that the job fails to finish within the user specified time constraint.

Table 1 shows the layout of the database. The information stored about each resource helps to predict the probability of job failure and predicted queuing time.

Resource_name	Text	The resource name
Total_started_job_requested	Number	Number of jobs execute from start on this resource.
Total_started_job_failed	Number	Total number of jobs, which started on this resource but failed to finish on time.
total_queuing_time	Number	Total of jobs queuing time
Queuing_time_standard_dev	Number	The Standard Deviation for Queuing time.

Table 4.1: Historical database layout

4.4.1 Predicting Job Failure

Grids are most formed with resources owned by many organisations, and thus are not dedicated for certain users. Thus, jobs that are dispatched to a remote site can possibly experience some reliability problems. The historical database contains information about resources than can be used to calculate the probability of job failure. The resource selector evaluates the candidate resource’s behaviour to choose the one most likely finish the job on time. As shown in Table 4.1, the resource selector considers the fraction of jobs failed as follows: $J_{failed} = \frac{number_{failed}}{number_{requested}}$, where the $number_{failed}$ and $number_{requested}$ are the jobs submitted through adaptive resource broker. The reliability is $R = 1 - J_{failed}$. This equation gives a simple evaluation of a candidate resource. The reliability percentage is defined as the Reliability Level (RL), which is the level of reliability resource can offer to the job.

The user should have the ability to specify a minimum level of reliability (e.g. 0.5), and thus the jobs are allocated only to those sites that can definitely satisfy the reliability requirement from the user, this is called reliability mode. Specifically, the user can specify the minimum acceptable value of reliability R ($0 \le R \le 1$).

4.4.2 Predicting Queuing Time

Predicting queuing time is another benefit from a historical database, as the resource broker submits the job to the local scheduler which places the job in a batch queue while waiting for execution. To estimate the amount of time a job has to wait in a batch queue before starting to execute is done by using historical data.

4.4.3 Survey of Predicting Queuing Time

The algorithms used in the local batch system schedulers are complex. In [104] explicit knowledge of the scheduling algorithm is used by the batch scheduler. While the algorithm may be known, the specific instance of the algorithm and the definition of any parameters it requires are the prerogative of the site administrators. A survey of the reservation capability of different batch systems can be found in [105] showing that not all batch schedulers support advance reservations. An alternative to advance reservation is to use the job start time estimation tool that is available in some schedulers, e.g. Maui [106].

4.4.4 The Role of Historical Database

Calculating accurate queuing time is a very hard task in the Grid environment. The only way to obtain queuing time is by contacting the local scheduler. Since not all scheduler have the same policy, studying the behavior of the scheduler can give a prediction of how long the job will wait in the queue.

The resource broker circumvents this obstacle by using a historical database. The information about job queuing time in a given resource can be stored when the job is submitted. The information extracted from the database is the mean queuing time which is used by the resource selector and decision maker in case of migration. The decision maker bases its predictions of how long the job takes in the new resource only by the observed history of previous waiting times.

4.4.5 Updating Historical Database

The historical database is fully automated, which raises an issue of how it can be updated and how frequently. The information such as requested job execution time and number of jobs submitted can be updated prior to run-time. The dynamic data such as total completion time and number of jobs failed can be updated after each time the job finishes execution. Finally, in case of migrating the user’s job to another resource, the prediction model (see Section 4.8) needs the estimated transfer time and queuing time to predict how long the job will take in the new resource.

The historical database is a significant component in the architecture. It supports the adaptive resource broker, filtering all the resources that likely can not handle the user’s job, automates the discovery of the resource’s dynamic status, and stores the history profile of past performance of the resources.

4.5 File Transfer Time

In determining whether to migrate a job, the adaptive service needs to account for the time it takes to transfer required files to the new resource. In order to estimate the transfer time, the estimator must predict the time needed to transfer the job and the files required. In order to do this, the estimator must have an estimate of the expected available bandwidth between the resources involved in the job submission. For this purpose, the Network Weather Service (NWS)[53] is used. NWS is a distributed system that periodically monitors and dynamically forecasts the performance of various network and computational resources. NWS combines bandwidth measurements with statistical methods to make short-term forecasts of the available bandwidth. There exist many other tools for measuring bandwidth, e.g. netperf[107] and thrulay [108], but these do not generate forecasts.

In order to measure the latency between two computers, A and B, a NWS process on computer A sends a small message to a corresponding process on computer B. The process on B immediately replies to the process on A, with the process on A recording the round trip time. NWS approximates latency as half of this round trip time.

To estimate throughput, the NWS process on host A sends a large message to the corresponding process on host B, and the process on B sends a small acknowledgement message back to the process on A. The NWS process on computer A estimates the transmission time of the large message as the round trip time, less the latency estimate described above. Throughput is then estimated as the message size divided by estimated transmission time. NWS keeps a record of previous estimates of throughput and latency, which it uses to predict future resource availability using statistical modelling techniques.

A setup of the NWS includes at least three processes; a name server which hosts register, a memory server which stores measurements and generates forecasts, and finally, a sensor which performs the measurements. Quantities that can be measured include network latency and bandwidth. The sensors register with the name server and store their measurements on the memory host. The times required for user's job, input and output files transfer are both determined using Equation 4.1 [109].

$$MT_{transfer} = Latency + \frac{M_{size}}{B_{b/s}}$$

Equation 4.1: File Transfer Time using the latency and bandwidth from NWS.

The equation depicts the predicted message transfer time ($MT_{transfer}$) as combination of latency and bandwidth, where M_{size} is the message size, $B_{b/s}$ is the number of bytes per second and $Latency$ is the start-up time, the time taken to reach destination, and any processing time at the other end. However, download of the output files is similar to upload of the input except in one important aspect. The size of the output files are not known in advance as these files are created during the execution of the job. Users often have some sense of the amount of output data their job is likely to generate. The estimate of the size of the various output files should be provided by the user and added to the total execution time.

4.6 Estimating Execution Time

It is assumed the user has provided both a reference execution time, so that a prediction model can be used to provide a dynamic estimate of the expected job completion time, and a maximum acceptable execution time. Previous efforts of application runtime estimation can be broadly classified in three categories:

- Code analysis [110] techniques estimate execution by analysing the source code of the task.
- Analytic benchmarking/Code Profiling [111] define a number of primitive code types. On each machine, benchmarks are obtained which determine the performance of the machine for each code type. The analytic benchmarking data and the code profiling data are then combined to produce an execution time estimate.
- Statistical prediction [112] algorithms make predictions using past observations. Another notable effort is that of Grinstein [113] who devised a technique that aims at predicting the behaviour of ATLAS applications. Their technique falls into the analytic benchmarking/code profiling category.

The technique used in this research to obtain the reference execution time is by running the job several times, then calculating the mean value of the reference execution time and the wall time. This is similar to [114] by using the previous run to predict the reference execution time. However, the prediction model (Section 4.8.3) assumes the user has a reasonable approximation of the application execution time prior to submission.

4.7 Job Submission

Prior to submitting the job, the resource broker uses GridFTP [36] as the data transfer protocol for all files. The input to the GridFTP is the locations of the files and the name of the resources.

As indicated in Section 3.3.7, the job submitted uses RSL syntax. After the resource has been selected and all the files transferred, the job can be submitted, as shown in Figure 4.3.

```
rsl="&(executable=/tmp/xto.sh)" +
             “count =1”+
              "(stdout=" + gassURL + "/dev/stdout)";
String contact =  nodeSelected ;
```

Figure 4.3: Resource Specification Language (RSL) Used to Submit Job

Figure 4.3 depicts the RSL that submits the job to the resource. The resource name has been defined in the variable nodeSelected. The rest of the RSL has been explained in Section 2.3.6. The Globus toolkit is used for job submission as Grid middleware, and this includes initialising the Grid certificate and the use of Java CoG kit[115]. Java CoG provides a Globus API in pure Java including the GSI, using the IAIK Java SSL libraries to delegate credentials. The CoG kit provides APIs for submitting jobs to Globus job-manager or SGE job-manager, transferring files using GridFTP implemented in Java and querying LDAP servers using the Java Naming and Directory Interface (JNDI). This research only submits the job to the local scheduler, not the Globus GRAM (fork).

The executable file, as shown in Figure 4.3, runs using shell script file. The shell script is a series of commands written in plain text.

4.8 Adaptive Service

The adaptive service is implemented with the objective discussed in section 1.5 in mind (e.g. separation of concerns and localisation). As shown in Figure 4.1, the adaptive service comprises of decision manager, monitoring tool, and migration engine. The monitoring collects the necessary information for the decision manager. The decision manger uses the prediction model to decide whether to migrate the job or not. However, in order to predict application execution completion time, the following are required:

- A reference execution time (either $T_{100\%}$ or something that enables this to be calculated).
- Monitoring information that enables CPU time spent on the job in the time interval (t_0, t_i) ($T_{CPU}(t_0, t_i)$) to be estimated.
- A reliable estimate of future average CPU usage, i.e. the mean CPU usage in the time interval ($\overline{F}(t_i, T_c + t_i)$).

The first is assumed to have been supplied by the user. The second is obtained by regularly sampling the CPU time spent on the job (since the previous sample) and using this to estimate the current and mean CPU usage. The next section describes how to collect the necessary information then using this information to predict application execution completion time. Following, the decision of whether to migrate the job to new resource and the migration process.

4.8.1 Monitoring the Application

The monitoring tool runs on the same resource as the user's job, and the information provided by the monitor is used by the adaptive service in real-time. Hence, the information is not stored in files. However, for future work the monitoring information would be stored in the historical database to study the performance behaviour of the resource. It is important to note that prior to the start of monitoring, the Process Identification Number (PID) is obtained, which is used by monitoring tool to identify the process corresponding to the user's job.

The monitor tool has two basic components as described in Chapter 3. The monitoring tool frequently measures the CPU time and wall time. These times interpreted for use by prediction model. The implementation of the components is discussed are below:

Sensor: The sensor measures the characteristic of the resource by directly reading the data from the operating system. In the UNIX operating system, there are several ways used to measure the usage of CPU by a specific application, for example using one of the commands ps or top. However, using the **PS** or **TOP** commands is not very accurate. By sampling the CPU at time interval (e.g. 5 second) using a command, such as **TOP** (see Figure 4 "using **PS** for mean"), the sensor ignores the values between the sampling, as a result the data may distant the estimated value for mean CPU usage.

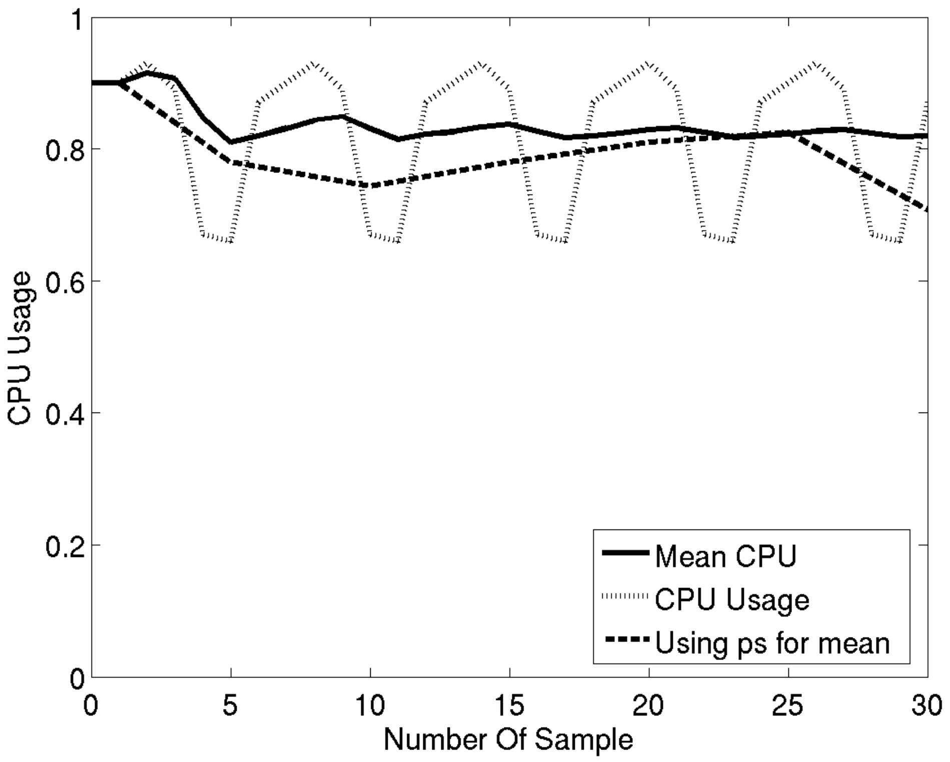

Figure 4.4: Sampling CPU Usage and Mean CPU using PS and PROC Commands

The implementation of the sensor takes into account the main characteristics described in Section 3.4.1 such as accuracy and locality. The sensor uses the data provided by the operating system which is located in **/PROC** directory. The **/PROC** pseudo file system is a real time, memory resident file system that tracks the processes running on the machine and the state of the system [116]. The sensor retrieves the user's job execution time (the amount of time the CPU spent on the job) and the wall time. As depicted in Figure 4.4, the "mean CPU" is generated from the data retrieved from the **/PROC** directory which shows more accurately than the "using **PS** for mean" CPU generated by **PS** command.

Interpreter: The information provided by the sensor is interpreted for use by the decision manager. The information, such as the job's execution time and wall time, are used to calculate the current and the mean CPU usage, as shown in equations 4.2 and 4.3, which provides the necessary information for the prediction model (explained in section 4.8.5).

$$\overline{CPU}\% = {T_{totalCPU}}\big/{T_{totalWall}}$$

Equation 4.2: Calculating Mean CPU

$$CPU_{current\%} = {T_{currentCPU}}\big/{T_{\text{int}\,erval}}$$

Equation 4.3: Calculating Current CPU

4.8.2 Decision Manager

The decision manager is the main component in the adaptive service which receives the data from the monitoring tool. The decision manager determines the future action when the job requirements are violated. The decision is based on predicting when the user's job finishes and querying for new resources by contacting the resource broker. The core part of the decision manager is the prediction model developed for this research. The prediction model takes input from the user such as reference execution time (an estimate of how much CPU time the job consists of on a particular resource), and in case of migration, the queuing time and file transfer time from the historical database and NWS respectively. The rest of this section describes how the transfer time, queuing time and remaining execution time prediction are accounted for by the decision manager.

4.8.3 Prediction Model

In order to make use of the adaptive service, it is clearly necessary to have a decision-making algorithm that initiates a search for alternative resources when the performance degrades to an unacceptable level. The inputs for the prediction model are file transfer time, predicted queuing time, execution time, and maximum acceptable execution time.

The prediction method used here is based on the pattern of CPU usage during run-time, and is therefore expected to display a high level of accuracy only for CPU-intensive applications.

Let F be the fractional CPU usage for a single application, and let $T_{100\%}$ be the time it takes for the application to complete if $F = 1$ for the entire duration. If the application begins running at time $t = t_0$, then at time t_i, the remaining CPU time of the job is,

$$T_{CPU\ remaining} = T_{100\%} - T_{CPU}\left(t_0, t_i\right)$$

Equation 4.4: Remaining CPU time of the job.

where $T_{CPU}\left(t_0, t_i\right)$ is the CPU time spent on the job in the time interval $\left(t_0, t_i\right)$. In our experiments, the resources used are all Unix machines, and the CPU time is found by picking out the system and user time from the appropriate files in the proc directory. The CPU time completed after time t_i is

$$T_{CPU}\left(t_i, T_c + t_i\right) = \overline{F}\left(t_i, T_c + t_i\right) T_c$$

Equation 4.5: CPU time completed after time t_i.

Here, T_c is the time elapsed since time t_i, and $\overline{F}\left(t_i, T_c + t_i\right)$ is the mean CPU usage in the time interval $\left(t_i, T_C + t_i\right)$. The time remaining is estimated by setting equation (4.4) equal to the right-hand side of equation (4.5) and solving for T_c , so that,

$$T_{remaining} = \frac{T_{100\%} - T_{CPU}\left(t_0, t_i\right)}{\overline{F}\left(t_i, T_C + t_i\right)}$$

Equation 4.6: User's Application time remaining

Equation 4.6 uses the input from the user ($T_{100\%}$) and the monitoring tool ($T_{CPU}\left(t_0, t_i\right)$ and $\overline{F}\left(t_i, T_c + t_i\right)$).

However, it is then assumed that the mean is an accurate estimator for future CPU usage. However, suppose that, over *N* successive samples, the current CPU usage value is less than the mean, or alternatively, over *N* successive samples, the current CPU usage value is greater than the mean. If *N* is sufficiently large, this would indicate that the mean is no longer a good predictor.

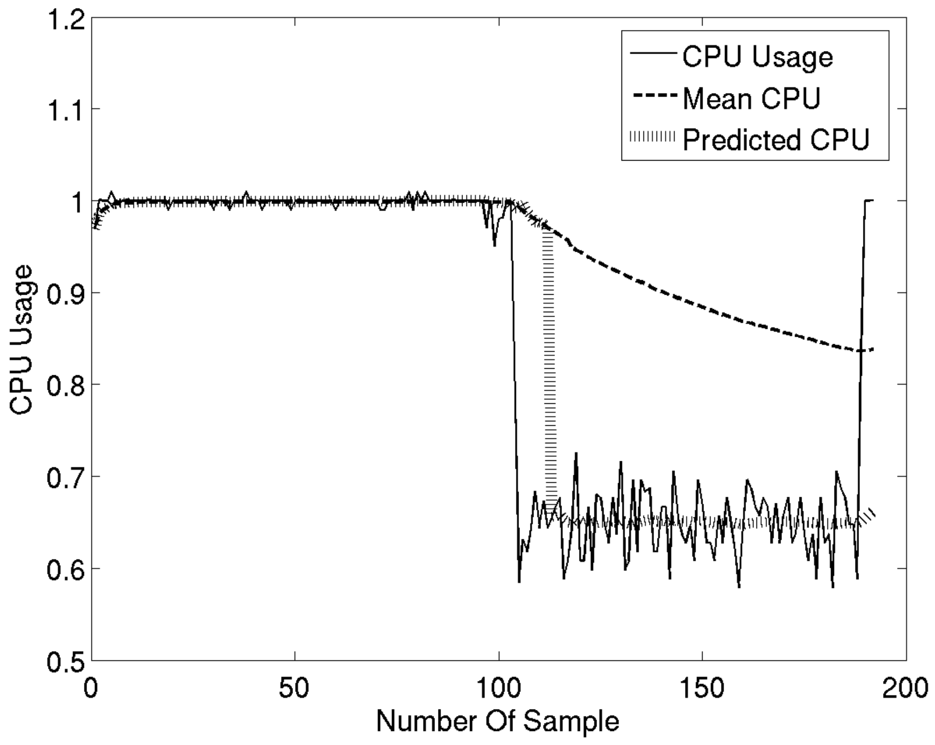

Figure 4.5: Predicted CPU vs. Mean CPU

This would happen as depicted in Figure 4.5, for example if during execution another application begins executing using the same CPU. In this case, it is better to use the mean CPU usage only over recent samples to estimate the future CPU usage. Note that choosing too large a value for N would result in performance degradation taking too long to identify, whereas too small a value could result in initiating migration unnecessarily. This is discussed further in the experimental section.

4.8.4 Decision Maker

The decision whether to migrate or not is based on the output of the prediction model ($T_{remaining}$, $T_{remaining(NEW)}$ and $T_{CPUremaining(NEW)}$), historical database ($T_{elapsed}$) and the user (T_{user}).

If the prediction indicates that the application will not finish within the specified time, then the progress of the job is saved, through the use of check-pointing and the resource selector is instructed to find an alternative resource. The job is migrated if a resource is found, satisfying the following condition:

$$T_{remaining(NEW)} + Ov < T_{user} - T_{elapsed}$$

Equation 4.7: Migrating decision.

Here, $T_{remaining(NEW)}$ is the predicted remaining time on the alternative candidate resource, where data from the monitoring tools is used to estimate the CPU usage. T_{user} is the maximum acceptable execution time specified by the user, $T_{elapsed}$ is the time elapsed, and Ov is the overhead associated with migrating the job (file transfer time, resource selection, job start-up, etc.).

Suppose a job is to be migrated to an alternative resource at time t_i, after it has partially completed execution that began at time t_0. Let the execution time at 100% CPU usage on the new resource be $T_{100\%}^{M} = YT_{100\%}$, where $Y > 0$. In that case, the remaining CPU time, if the job completes on the new resource, is given, using equation (4.4), by

$$T_{CPUremaining(NEW)} = YT_{CPU\ remaining} = T_{100\%}^{M} - YT_{CPU}\left(t_0, t_i\right)$$

Equation 4.8: Predicting completion time in the new resource.

If it is necessary to restart the job from the beginning then $T_{CPUremaining(NEW)} = T_{100\%}^{M}$.

In addition, it is assumed that the job will need to wait in a queue at the new resource for (on average) time $\overline{T}_q^{M}$ and that transfer of all required files will result in an additional overhead, $\overline{T}_{transfer}$. Hence,

$$Ov = \overline{T}_q^{M} + \overline{T}_{transfer}$$

Equation 4.9: Estimating overhead in the new resource.

4.8.5 Migration

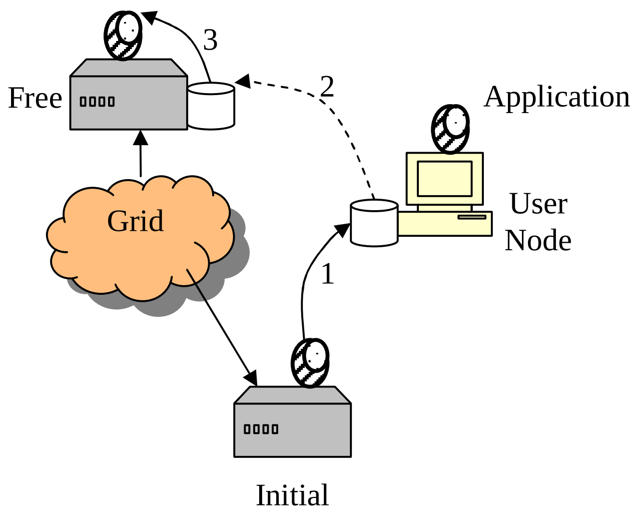

Figure 4.6: Migration process

Migration is one of the key functionalities needed for implementing an adaptive service. In Grid environments, checkpointing is necessary for enabling other features such as job migration.

The migration component is responsible for saving the state of the job, stopping the job, and restarting it, as shown in Figure 4.6. Migration is only performed when the decision manager decides. Hence, the job needs to be checkpointed, terminated, and restarted.

When the job needs to be migrated, the checkpointing process saves the state of the job and transfers it to the user's node (step 1 in Figure.4.6). Since the language used is Java, saving the state of the job is as simple as writing the user's application to a file. The main advantage of saving the job in different node is in the case if machine crashes, then the job can be rolled back as last saved.

The migration process terminates the job and waits until the resource broker transfer all the files to the new resource. The resource broker transfers the files, including the file image of the job (2 in Figure 4.6). The migration tool makes sure the job starts from where it stopped. This can be done by writing the saved file to the user’s job (3 on Figure 4.6).

4.9 Summary

This chapter described the implementation part of the adaptive service and the part of the resource broker involved in the adaptivity. The historical database plays an important role for selecting the most reliable resource. Also, the historical database provides prediction for queuing time which is important in the case of migration.

This chapter shows the flow from writing the code through binding process, selecting resources, job submission, and finally adaptive service.

The adaptive service includes monitoring tool, decision manager, and migration engine. The monitor uses NWS to obtain forecast of the network. The forecast helps to predict the file transfer time between resources. The transfer time and queuing time are important factors in deciding to migrate.

The heart of the adaptive service is the prediction module which provides information for the decision maker to make proper decision. The prediction module takes the input from the monitor and the resource broker.

Migration responsibility is to checkpoint the job, stop the job, and then restart it in the new resource. Checkpointing saves the state of the job in order to transfer it to the user’s node to restart the job from the point it stopped.

I find that the harder I work, the more luck I seem to have
— Thomas Jefferson

Chapter 5

Evaluation and Discussion of Adaptivity in Grid Resource Broker

5.1 Introduction

In this chapter evaluation of the adaptivity in the resource broker will be presented. For this research, the experiments went into two stages. The first stage was using test-bed machines which is an environment used for developing, testing and evaluating Grid technology research. There are issues raised and lesson learned from the first stage which is have been considered in the second stage. The second stage was using White Rose Grid (WRG). The White Rose Grid project operates under the auspices of the White Rose University Consortium, which is an affiliation of the three Yorkshire Universities of Leeds, York and Sheffield [17]. The WRG operates as a Virtual Organisation with dedicated resources in each institution.

This chapter is organised as follows, first is an overview and objective about the experiments, then discussion and evaluation of test-bed experiments. There follows a list of issues raised and lessons learned by running the experiments on test-bed. Further experiments will be discussed and evaluated in the WRG after considering the issues raised and lessons learned. Finally, a study of the overhead of the adaptive service has been carried out.

5.2 Overview of the Experiments and Objectives

As discussed in section 3.4, the adaptive service is expected to provide a performance enhancement to the user's job in terms of the time interval between submission (to the broker) of the user's job requirements and the job finishing execution. In particular the vision of prediction model used at run-time is expected to provide an enhancement by ensuring that decisions are made on the basis of accurate information. Specifically, the experiments involve a performance comparison of broker with adaptive service, compared to the broker without adaptive service on two different environments, the Grid test-bed and then the White Rose Grid (WRG).

The experiments carried out on both environments are designed on the basis of two common objectives which are:

1. What is the impact on execution time of sharing CPU between the application and the adaptive service?
2. Does the adaptive service provide a performance enhancement, in terms of job execution time, in the event of resource performance degradation?

The experiments were firstly carried out on a local Grid test-bed as it is an environment used for developing, testing and evaluating Grid technology research. These experiments were then carried out on the WRG, a large distributed Grid infrastructure (further described in Section 5.4) which spans across three administrative sites and hold true Grid attributes described in Section 2.1, such as site autonomy and heterogeneous substrate. Even though the same experiments were carried out on both environments the parameters boundaries differed due to the different infrastructure size. This will be discussed when describing the experiments for each environment. Further, the DAME XTO jobs and calculating π application are used for the experiments.

5.3 Experiments on the Grid Test-bed

The experiments were performed on a Grid test-bed consisting of 10 machines. Each machine has a Pentium IV processor (1.2GHz) with 256 Mb RAM. The operating system is Linux 2.4. Globus 2.4 and Sun Grid Engine 5.3 are installed on all machines.

There is a Grid Resource Information Service (GRIS) associated with each of the machines. Communication occurs with a fast (100Mb/s) LAN network. The network overhead is not addressed when running the experiments in test-bed. Thus, the file transfer and queuing time are not accountable for the prediction model as the test-bed machines are isolated from the outside world. Hence, the machines are dedicated to very few users.

5.3.1 Experimental Design

The experiments carried out can be described in terms of the following scenarios:

The user specifies the job (let this be job *A*) requirements (e.g. the time needed to execute the job). After the resources have been located and the job has been dispatched to the selected resources, the adaptive middleware monitors the resources and ensures the requirements are satisfied. Suppose another user accesses a resource being used for job *A* and begins running another job (*B*). Since job *B* uses some portion of the resource, job *A* takes longer to finish. In this case, the monitor passes information to the decision manager, based on the prediction formula and the decision manager initiates migration to ensure the job continues to run on resources that meet the minimum job requirements.

The scenario above provides a setting in which looking into providing adaptivity to user's job can be investigated. The experiments presented involve the submission of jobs, with user requirements specified by the user, to the resource broker. While a job is running, other jobs may be submitted to the same resource(s). The results obtained are compared to the case where the adaptive service is not used. In particular, the experiments address the following questions:

- When job requirements are not met, are jobs being successfully migrated?
- Does this result in shorter job execution time, compared to the case when the adaptive service is not used?

In reference to Section 4.6.2, note that the experiments running in test-bed are not considering the transfer time and the queuing time in the decision maker. Also, the main

CPU usage was considered to be an estimate for future CPU usage. Moreover, the samples are taken in fixed time interval which is 5 seconds.

Prior to running the experiments, the user job is run and the CPU usage periodically measured. This is done so that a value for $T_{100\%}$ (in prediction model Section 4.8.3) can be obtained. Specifically, this is to enable the experiments to run with the assumption that the user knows how long the job would take to run with 100% (or some other percentage value, enabling $T_{100\%}$ to be calculated) CPU usage. To obtain $T_{100\%}$, the user's job has been executed several times on a dedicated resource. Hence, no other job is running simultaneously with user's job. However, the chosen job was run and took 4469 seconds, with a mean CPU usage of 0.98, resulting in a value of 4380 seconds (73 minutes) for $T_{100\%}$.

Two experiments are then run:

1. *First scenario*, the job is executed from start to finish and the remaining job time predicted during the course of the computation is compared to the actual execution time.
2. *Second scenario,* the job is executed and during execution, other jobs are submitted to the same resource. This initiates a migration. This is compared to the case where adaptation is not used.

For the first scenario, experiment is used to assess the validity of the prediction formula. This involves running the user's job without interference of any other job running in the same resource. Hence, the resource is dedicated to the user's job. For this experiment, the job is monitored but no migration or any decision will be taken. However, the prediction model is used to calculate the remaining time.

For the second scenario, an experiment is used to assess the effectiveness of adaptation in ensuring timely job completion. The prediction model is validated and is used for production in this experiment. When the job starts executing in the resource, third party job starts few second later. However, the decision maker is slightly different from the one described in Section 4.8.4. The decision maker migrates the job under the following condition only:

$$T_{remaining} \geq T_{user} - T_{elapsed}$$

OR

$\overline{F}$ decreases for 10 successive samples and

$T_{worst-case} \geq T_{user} - T_{elapsed}$

Here T_{user} is the user specified execution time and $T_{elapsed}$ is the time elapsed since the job began execution. The worst-case estimate of $\overline{F}$ is obtained by taking the mean value of the CPU usage over the 10 samples since it began to decrease. Thereafter, $\overline{F}$ is calculated from this point in the computation. Essentially, this is addressing the fact that the CPU usage has dropped and it may remain at its new level for the remainder of the computation. The choice of 10 samples in condition 2 is somewhat arbitrary: the larger this value, the less likely it is that a job is unnecessarily migrated due to a fluctuation in CPU usage, while the smaller this value, the faster the need for migration can be identified. Further investigation is planned to identify an *optimal* number of samples.

5.3.2 Performance Results and Discussion

Experiment 1 was carried out, as described above. No additional jobs were submitted to the chosen resource during execution. Figure 5.1 shows the remaining time predicted by the formula plotted against the actual time remaining until the job completes. The latter is deduced retrospectively once the job has executed and the total run-time is known. The plot shows a good match between the predicted and the actual execution times, which enhances confidence in the validity of the predicted model.

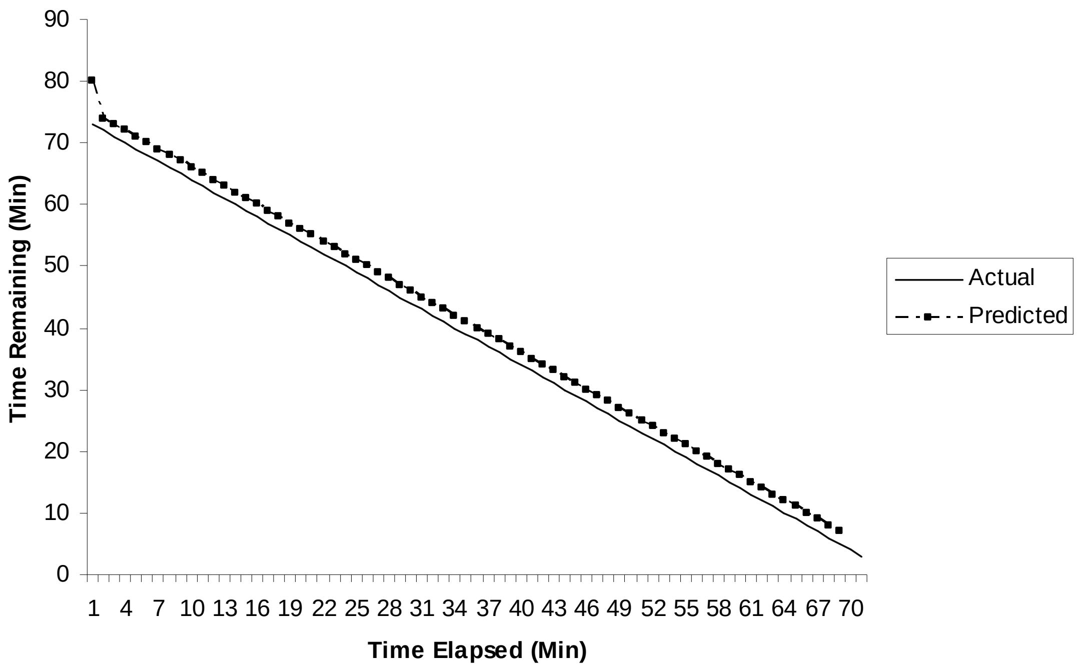

Figure 5.1: Comparison of remaining execution time with predicted time remaining

At the start of experiment 2, the user submits the job requirement, whereupon the resource discovery and selection layer decides which resource meets the requirements. Specifically the job requirement is a time-constraint: the job must be executed within 4867 seconds (81 minutes 7 seconds). This corresponds to the length of time the job would take at 90% CPU usage. This is justified by the need to choose an execution time that is not less than the execution time obtained in the run prior to the experiments. Otherwise the likelihood the user's requirements would be met would be low, even if no other jobs were submitted to the system.

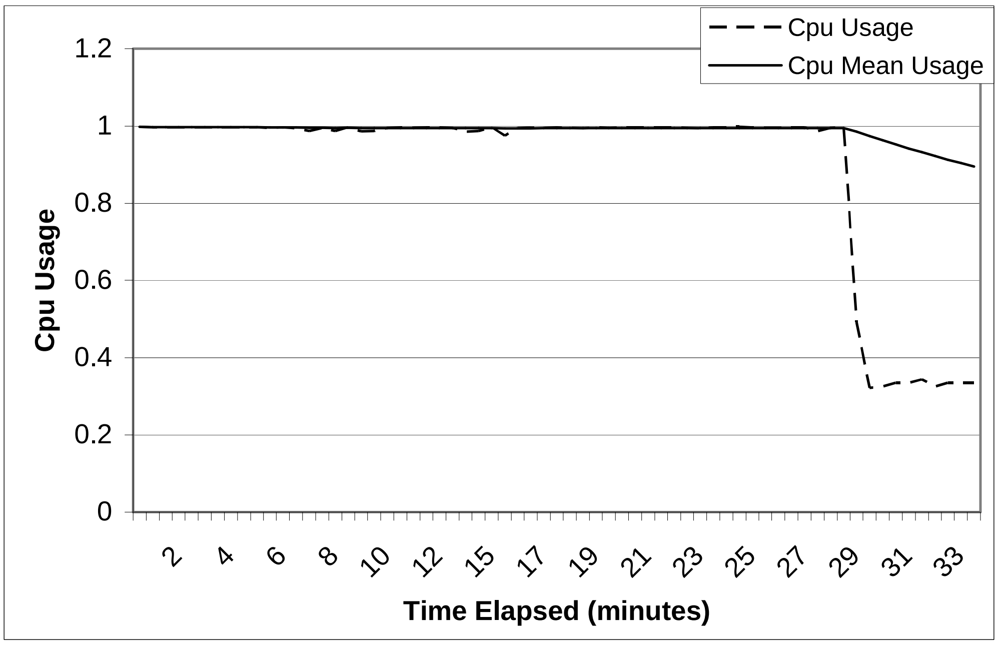

Figure 5.2: CPU usage during run-time in resource

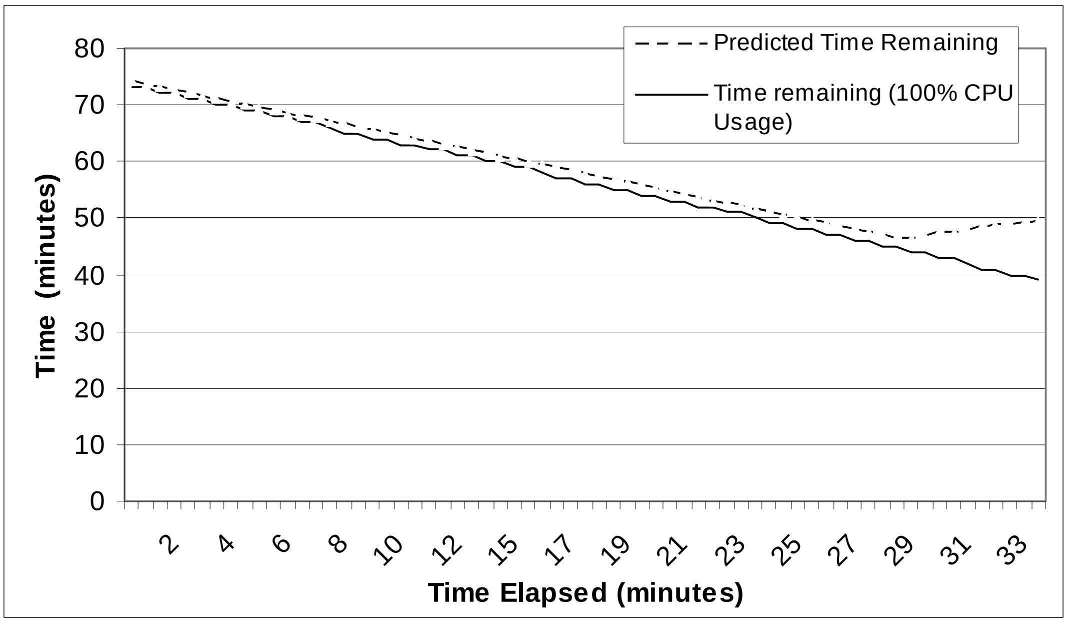

Figure 5.3: Predicted time remaining during run-time in resource 3 before migration

Figures 5.2 and 5.3 refer to the job running on resource 3. The job executes normally (i.e. with close to full CPU usage) for about 30 minutes. As indicated by the dotted line in Figure 5.3, which shows the predicted remaining time, the predicted remaining time begins to increase after this point. This indicates that the CPU usage has decreased sharply, as confirmed by the data shown in Figure 5.2. Hence the resource is no longer expected to finish the job within the specified time constraint. This means the broker has to take action and restart the job on another resource.

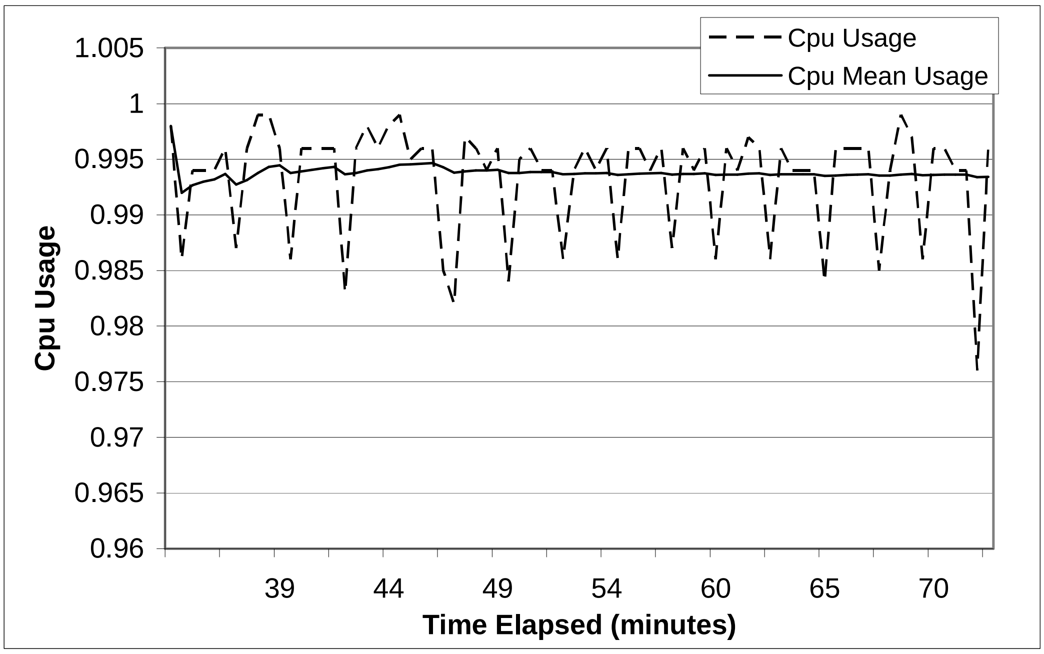

Figure 5.4: usage during run-time in resource 4

The broker has chosen resource 4 as shown in Figures 5.4 and 5.5. This shows that the job starts from where it stopped on resource 3. 34 minutes have elapsed since the job began execution. Hence, in order to meet the user requirement the job must complete within a further 47 minutes on resource 4. Shortly after the job continues execution on resource 4, the predicted model is used to predict the remaining time. Figure 5.5 shows that the job is expected to finish within the specified time. The total execution time on both resources was 74 minutes. As a consequence of the use of adaptive service, the job is adaptable to the changes of the environment and continues to meet the user requirements.

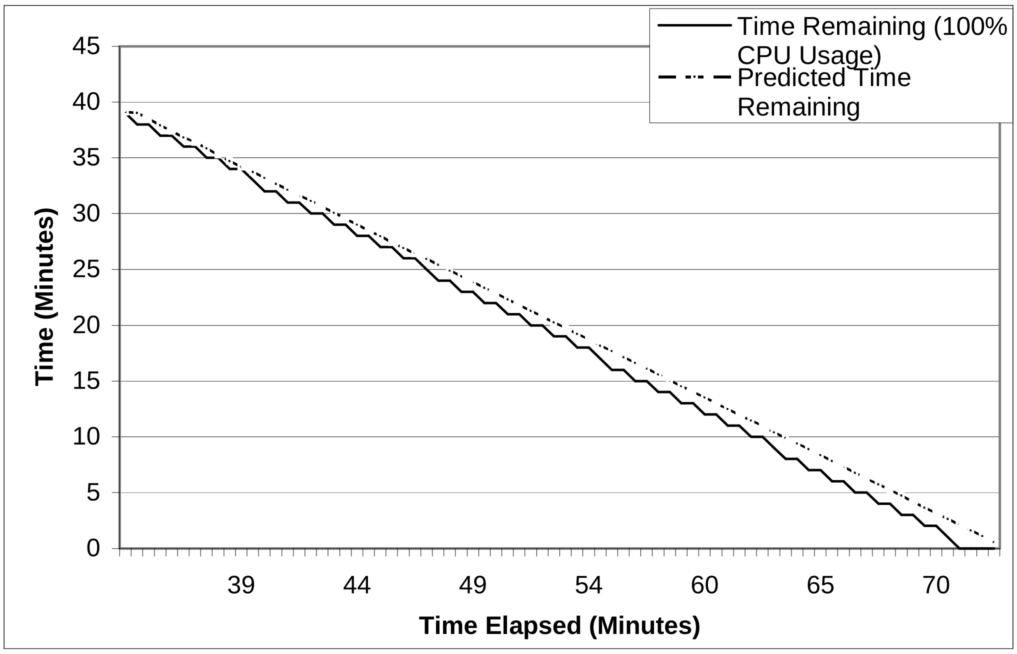

Figure 5.5: Predicted time remaining during run-time in resource 4 after migration

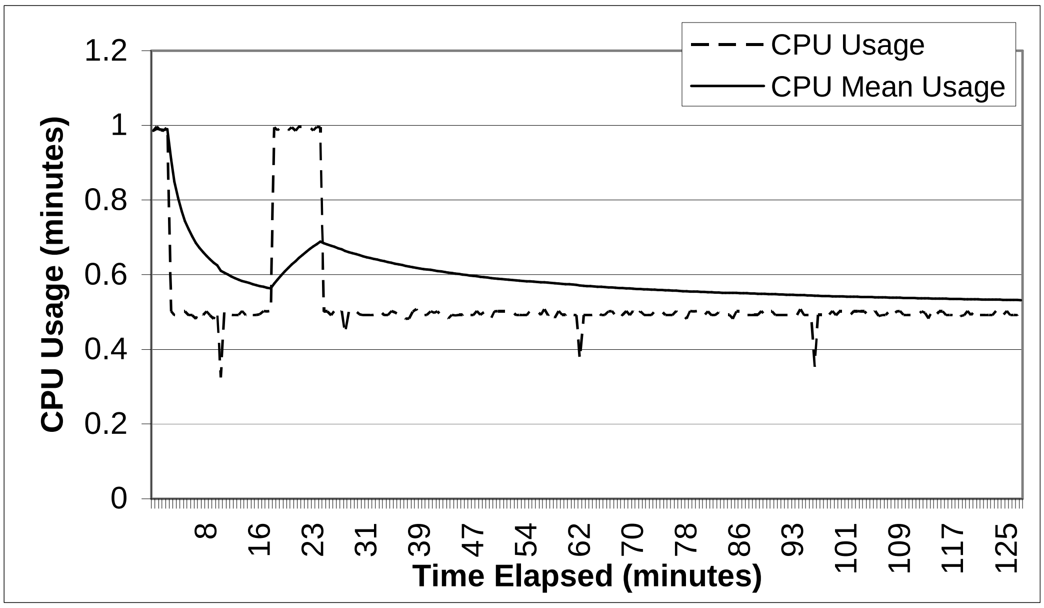

Figure 5.6: Job Running on Resource 6

The same job was executed without the adaptive service, as shown in Figure 5.6; in this case the job took longer than the time it was assigned, i.e. it ran for more than 7200 (125 minutes). This results from the fact that there was another job running on the same resource during the job execution.

Referring to the questions posed before in Section 5.2, it has been shown through experiments that the adaptive service is successfully supporting job migration. This results in a reduction in the job execution time compared to the case where non-adaptive service is used.

The experiments described in this section have shown that by using the reflective technique the user can be isolated from all the complexity of the system. Most importantly the adaptive middleware significantly increases the likelihood that a job can be executed within a time specified by the user.

5.4 Test-bed Experience

This section discusses the experience gained from running the experiments previously described. There are issues raised if running the same experiment on the WRG and lessons learned from the previous experiments.

5.4.1 Issues Raised

There are several issues raised from running the experiments on test-bed machines which were implemented next on the WRG. They are as follows:

- *Environment setup:* The test-bed machines exist in a single administrative domain, all with the same operating system, file system, etc. This is in contrast to the heterogeneous environment of a typical Grid, where additional measures are necessary to ensure the adaptive service works. A script is needed to set up the environment, specifying for example the location (on the executing resource) of the Globus directory and the directory the job is to be executed in.
- *Resource performance:* The test-bed is a small and closed environment which makes the behaviour of the resource known and easy to identify.

However, on a WRG environment, it is essential to build knowledge about Grid resources such as historical database.

- *Migration:* The test-bed machines are connected over a LAN. Consequently the migration overhead is small. In contrast, in a real Grid environment, migration may require transfer of large files over a large distance.
- *Checkpointing:* In the test-bed, which is in a single administrative domain additional software can be installed to support checkpointing. In a distributed system, with different operating systems and local administrative domains, this is more difficult. With a java application it is possible to take an image of the class file during run-time. However with non-java applications we assumed that if a job is to be migrated it must be re-started. An example of checkpointing in a UNIX environment such as MOSIX [117]. However, few local scheduler provides tools built on such as SGE checkpointing [23]. Incorporating a general checkpointing mechanism into our adaptive service is a topic of future research.
- *Reflective technique:* A further point is that on the WRG experiments discussed below, a real-life application is used rather than a simple open source application. In many cases, particularly in a commercial environment, the adaptive service will not have access to the source code. The reflective software used to bind the user's job with the adaptive service on the test-bed is openjava [118]. In the WRG experiments, the user's job is only available as an executable which makes it necessary to find reflective software that handles executable code. In this case, javassist [18] is used, as discussed in Section 4.3.

5.4.2 Lessons Learned

A lesson learned is the knowledge gained through experience or study during and after the experiments. It is a reflection on what was done right, what would be done differently, and how this could be more effective in the future. Taking the lesson learned into consideration and applying it to the experiments conducted on the WRG would enhance the adaptive service. However, a WRG experiment uses the mean CPU usage only over recent *(N)* samples to estimate the future CPU usage. Note that too large a value for *N* would result in performance degradation taking too long to identify, whereas too small a value could result in initiating migration unnecessarily. This is discussed further in the experimental section; the test-bed experiment uses the mean CPU as future prediction.

5.5 White Rose Grid Experiments

As mentioned before, the Grid test-bed is used for development as well as testing and evaluating Grid technology research. However, it is a local environment which does not fully include Grid resource management issues such as site autonomy, heterogeneous substrate and more importantly, the distribution of resources across different sites. The WRG (Figure 5.7) addresses these issues as it is an example of a computational Grid. WRG consists of resources at Leeds, York and Sheffield Universities. The approach to building the WRG has been to bring together the provision of High Performance Computing (HPC) services with emerging Grid technology (e.g. Globus and SGE). The high level objectives behind the formation of the WRG were as follows:

- to create and operate an enabling e-infrastructure, the Grid, that supports scientific collaboration across the three universities;
- to offer much larger pool of resources which can be easily selected by individual researchers to best match their projects' needs.
- To deliver stable HPC services at all three sites;
- To work with regional partners and businesses to gain the experience of delivering a stable Grid service.

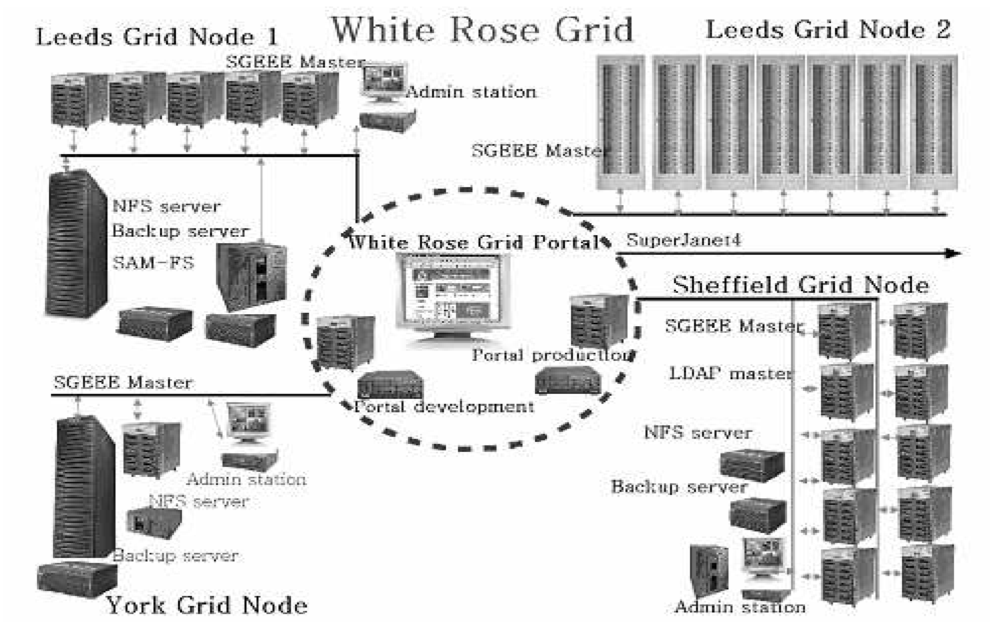

Figure 5.7: White Rose Grid (WRG) Architecture

The WRG is heterogeneous in terms of its underlying hardware and operating system. Two large compute nodes are situated in Leeds (Maxima and Snowdon), one at York (Pascali) and another at Sheffield (Titania). These nodes are connected by a fast network and offer significant heterogeneous computational facilities. Figure 5.7 shows the architecture of the WRG and depicts all machines at the various sites. The specification of these compute nodes is described below:

- Maxima (Leeds node 1): A constellation of shared-memory systems which include a Sun Fire 6800 with 20 UltraSPARC III Cu 900Mhz processors, and five Sun Fire V880 servers, each with 8 UltraSPARC III Cu 900Mhz processors.
- Snowdon (Leeds node 2): two Beowulf type clusters: a 28 processor development cluster and a 256 processor production service cluster; each using a Myrinet 2000 switch for interconnecting nodes. The production cluster supports both capability and capacity jobs. The main cluster comprises a head-node configured with a dual 2.2 GHz Intel Xeon and 256 processors configured out of dual processor nodes based on 2.2 and 2.4 GHz Intel Xeon processors.

- Titania (Sheffield node): the originally established node comprising a set of shared-memory servers. Iceberg - is a 160 processor Beowulf type system, supplied by Streamline Computing and Sun Microsystem. Titania was originally configured with 10 Sun Fire V880 servers, each with 8 UltraSPARC Cu processors. Iceberg is made up of a mixture of 2-way nodes each with 4GB memory and 4-way nodes; all nodes are based on AMD Opteron processors. The development cluster is of similar configuration.
- Pascali (York node): Fermata is a Sun Fire 6800 with 20 UltraSPARC III Cu 900Mhz processors, and Pascali is a Sun Fire V880 server with 8 Ultra SPARC III Cu 900Mhz processors. Nevada – a Beowulf type cluster based on Intel Pentium III processors.

- Maxima, Pascali and Titania are built from a combination of large symmetric memory Sun servers and storage backup running on Solaris, whereas Snowdon comprises a Linux/Intel based computer cluster connected with Myricom Myrinet. The middleware infrastructure is enabled through the use of Globus 2.4, while SGE handles the local job scheduling. All jobs must be routed through SGE for resource allocation on compute nodes within a particular cluster. To achieve this integration between Globus and SGE, the WRG has employed the Globus SGE job manager and information provider packages from Imperial College, London [119].

5.5.1 Experiments Design

Experiments have been carried out to address two key issues relating to the adaptive service.

- What is the impact on execution time of sharing CPU between the application and the adaptive service?
- Does the adaptive service provide a performance enhancement, in terms of job execution time, in the event of resource performance degradation?

The first question concerns the overhead of the adaptive service. The experiment has been discussed in Section 5.5.3. The main experiments are aimed at addressing the second question above and were carried out on the White Rose Grid, which consists of compute resources located at York, Leeds and Sheffield Universities.

The experiments are used to evaluate the behaviour of two applications using the adaptive service and compare this to the case where no adaptive service is used. Results are obtained for both a compute intensive application, which calculates π and the XTO (eXtract Tracked Orders) application: an aircraft engine diagnostic application which was used in the DAME (Distributed Aircraft Maintenance Environment) project [120]. In each case, provisional runs are used to obtain a reference execution time ($T_{100\%}$). The experiments are then carried out in three parts. Firstly the application is run a number of times to evaluate the accuracy of the run-time prediction used in job migration decision-making. Secondly the application is run and other jobs submitted to the same CPU during run-time. The timing is carefully controlled to ensure the experiments could be replicated. The performance is evaluated when no adaptive service is used. Finally, the second part is repeated but this time using the adaptive service.

The values of $T_{100\%}$ obtained for the White Rose Grid machines, Pascali (located at York University) and Snowdon (located at Leeds University) are shown in Table 5.1, along with the standard deviations. These were obtained by running the applications 75 times on each resource. Notice in table 5.1 that Snowdon machine runs slower than Pascali and dedicated lower CPU power, even there are no other jobs running in the same resource at the execution time. However, Snowdon dedicated lower CPU power than Pascali when running XTO but faster execution time, which means Snowdon is faster with I/O Jobs as XTO is mainly I/O intensive.

Machine Name	Application	$T_{100\%}$	Average Execution time/Standard Deviation/ Average CPU
Pascali Snowdon	Calculate π	486.66s 465.73s	487.93s/2.01s/ 95.7% 699.43s/4.14s/ 66.6%
Pascali Snowdon	XTO	655.22s 213.91s	659.92s/1.16s/ 99.3% 404.91s/50.55s/ 53.4%

Table 5.1. Experimental Parameter Values

5.5.2 Performance Results and Discussion

In the evaluation experiments below, the aim is to ensure the application completes execution within a user specified time, T_{user}. In these experiments, T_{user} is chosen to be 3 standard deviations larger than the average value of $\overline{T}_{total}$ (the average completion time, including waiting time in the queue, i.e. measured from submission until execution completion). Since the standard deviation for execution on Pascali (the machine the applications begin execution on, when evaluating the adaptive service) is small, this means that any performance degradation must be quickly identified and migration initiated or the user time constraint will not be met. The values of T_{user} are 503.8sec and 683.6sec for the π application and XTO respectively.

Monitoring of the application is performed at 2 second intervals and the value of *N* used here is 10. This corresponds to basing the CPU usage estimate on a moving average if the average value continually increases (or decreases) for a period of 20 seconds. This rather short time period is appropriate, since the execution times are quite short (for both applications) and, since the values of T_{user} are only a little larger than the average execution times, there is little margin for error. In other words, it is necessary to identify performance degradation quickly or migration will not ensure timely job completion. Further investigation is needed to evaluate the impact of varying *N* for different application run times.

The issue of estimating migration time has not been investigated in detail. For the purpose of these experiments the sample mean queuing time, along with standard deviation, has been obtained from the historical database. The data collected by performing the task more than 50 times. On the other hand, file transfer times for each application were obtained by contacting NWS. This information is then used by the job migration decision-making algorithm.

5.5.2.1 π Application

Our experiments on the π calculating application indicate that the run-time prediction is very accurate for this application when the CPU usage remains stable (i.e. no other jobs are submitted to the same CPU, the machine doesn’t crash, etc.). Figures 5.8 and 5.9 are showing the results for Snowdon and Pascali. In both cases, the predicted remaining time is plotted alongside the actual remaining time. The actual remaining time is only known after the application has completed. However, including this on these plots highlights the accuracy of the prediction and also enables the efficiency with which the prediction is adjusted when resource performance degrades. This can be seen by examining Figure 5.10 and 5.11, which shows (for Snowdon and Pascali) the predicted remaining time, when two additional jobs are submitted to run on the same CPU approximately 30 seconds after execution begins. Note that the prediction (in both cases) quickly adjusts to account for the reduction in CPU usage. On both Pascali and Snowdon, the execution time is far greater than T_{user} when the adaptive service is not used.

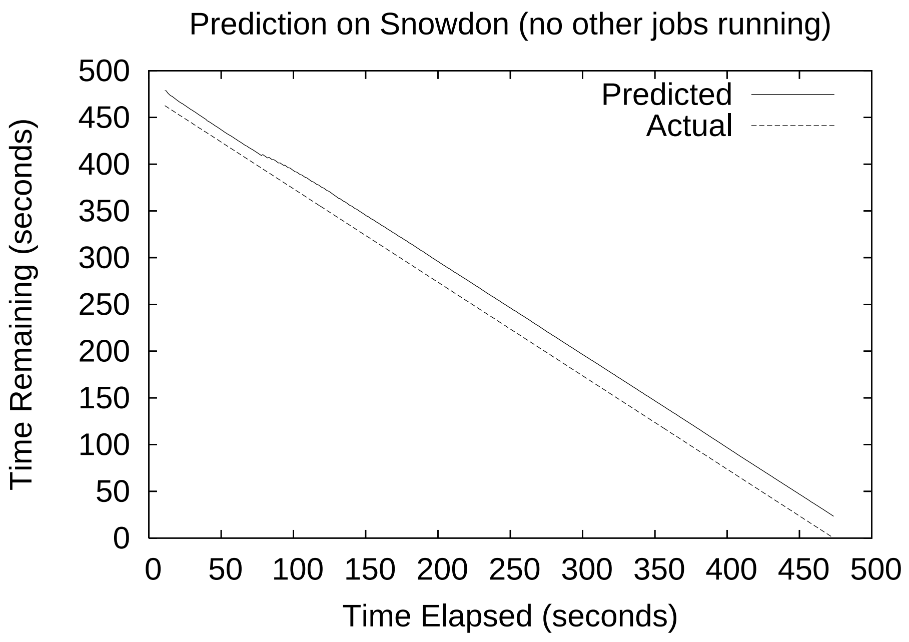

Figure 5.8: Prediction Evaluation on Snowdon (no other jobs running)

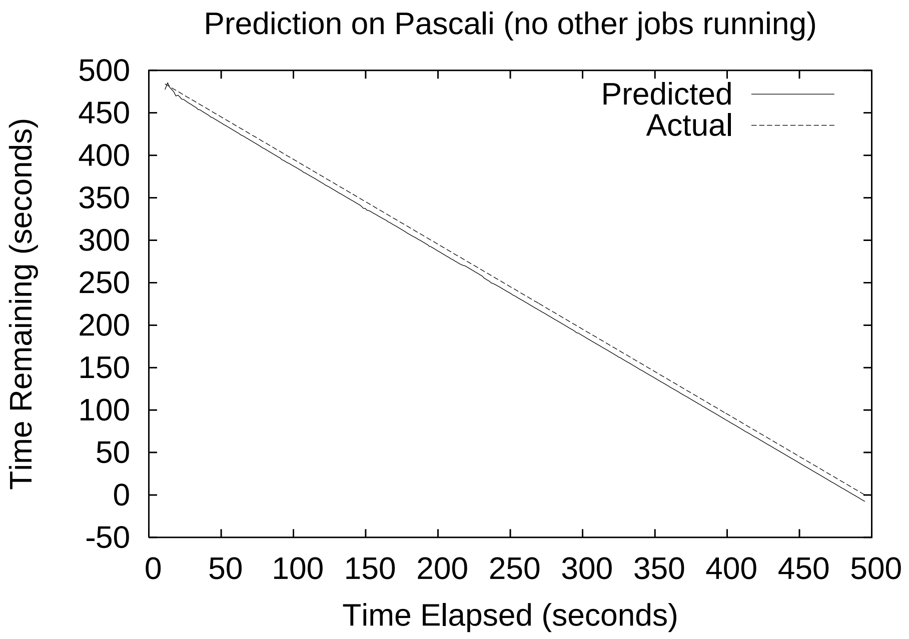

Figure 5.9: Prediction Evaluation on Pascali (no other jobs running)

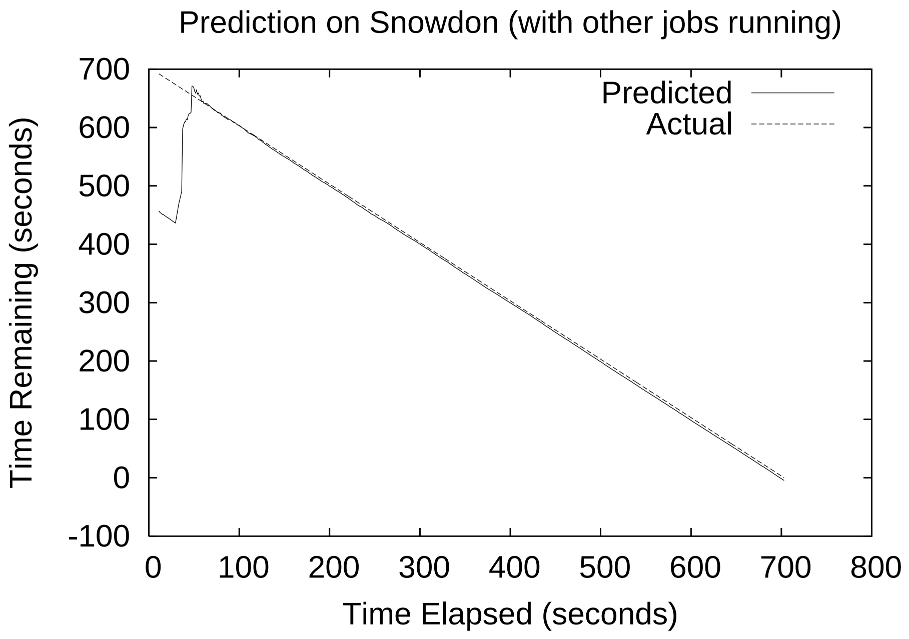

Figure 5.10: Prediction Evaluation including resource performance degradation (Snowdon)

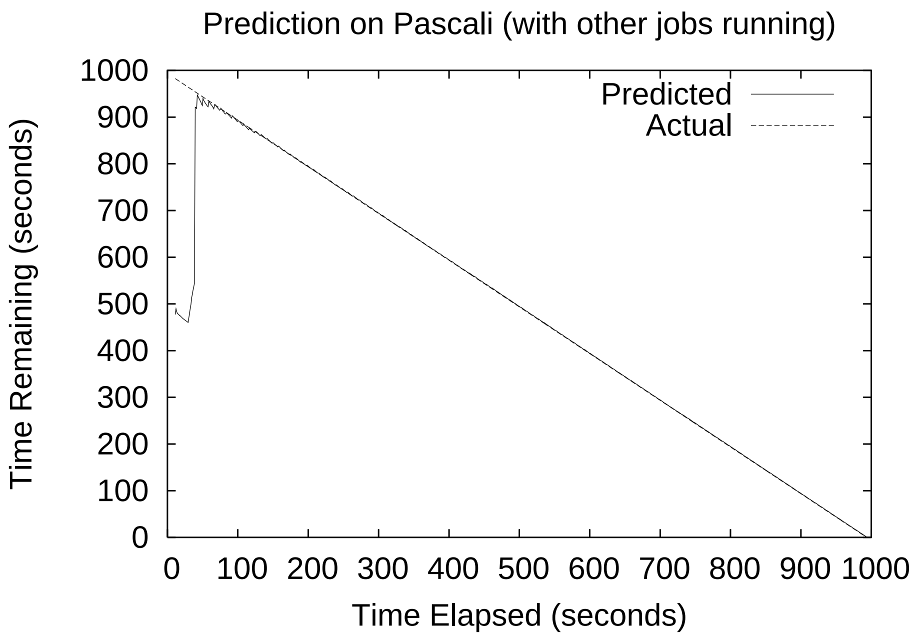

Figure 5.11: Prediction Evaluation including resource performance degradation (Pascali)

Figure 5.12 and 5.13 shows the results obtained in evaluating the effect of the adaptive service on performance, under the same conditions as the results presented in Figure 5.10 and 5.11 were obtained. The first plot shows the predicted remaining time on Pascali. The prediction algorithm quickly adjusts to account for the performance degradation and once the sum of the predicted remaining time and the elapsed time exceed the user-specified time constraint (and the completion time if the job is migrated is predicted to be earlier), the job is migrated to Snowdon. The second plot shows the predicted remaining time on Snowdon until the job finishes executing. In this case, while the adaptive service resulted in a substantial speed-up over not using the adaptive service, the user's time constraint was still not met. The total execution time (including queuing and file transfer times and migration overhead) is 523.72 seconds. This results from the fact that the time taken to transfer files is significantly greater than the difference between the average execution time on either resource and T_{user}. Nevertheless the speed-up obtained by using the adaptive service is significant.

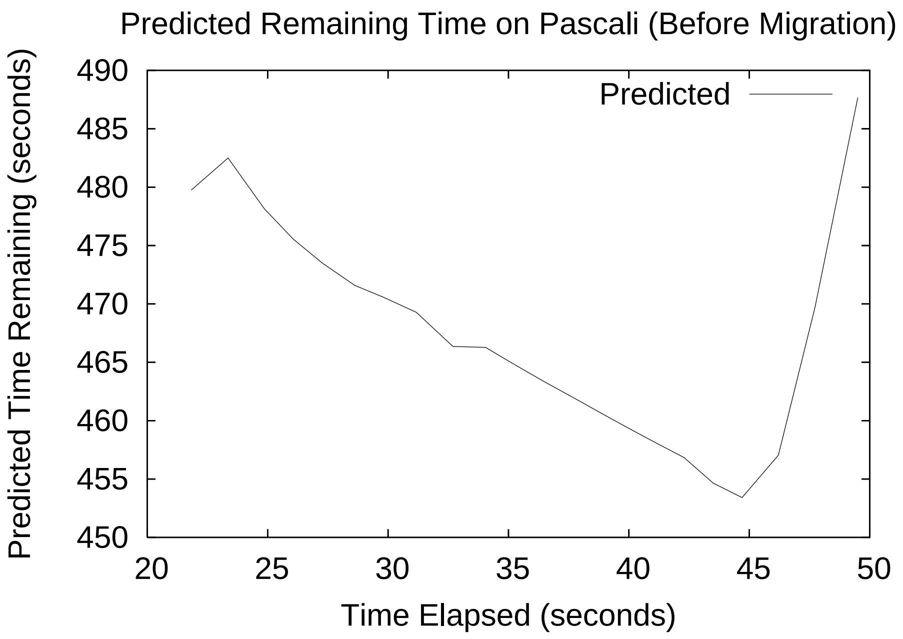

Figure 5.12: .Predicted remaining time prior to migration

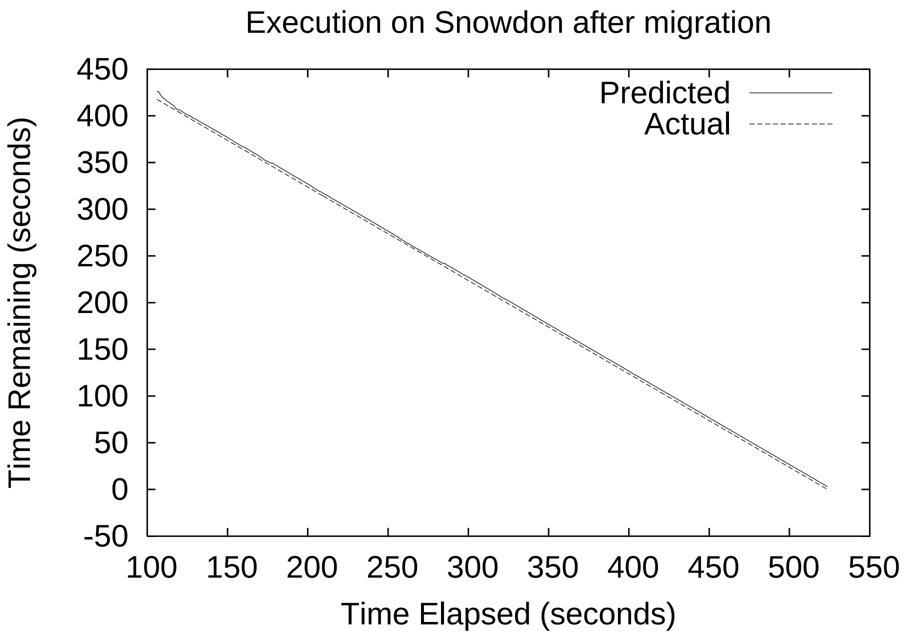

Figure 5.13: remaining time on Snowdon after migration

5.5.2.2 XTO Application

Figures 5.14, 5.15, 5.16, 5.17, 5.18, and 5.19 show the analogous results for the XTO application. On both machines, there are clear discrepancies between the predicted and actual remaining time near the beginning of execution. The main reason for this is that there were significant fluctuations in CPU usage in the first few seconds of execution. In this case (see Figure 5.16 and 5.17), the use of the adaptive service again results in significantly faster execution (execution on Pascali took almost 2000 seconds when other jobs were submitted to the same resource without the adaptive service (see Figure 5.18 and 5.19) and ensure that the user time constraint of 683.6 seconds is met. The total execution time (including queuing and file transfer times and migration overhead) is 586.66 seconds. Notice in Figures 5.14 and 5.16, the remaining and elapsed time in both are the same due to the fact that XTO application is I/O intensive. Therefore, running another application during the execution of XTO will not have an impact on the predicted and actual execution time as XTO takes small fraction of CPU load.

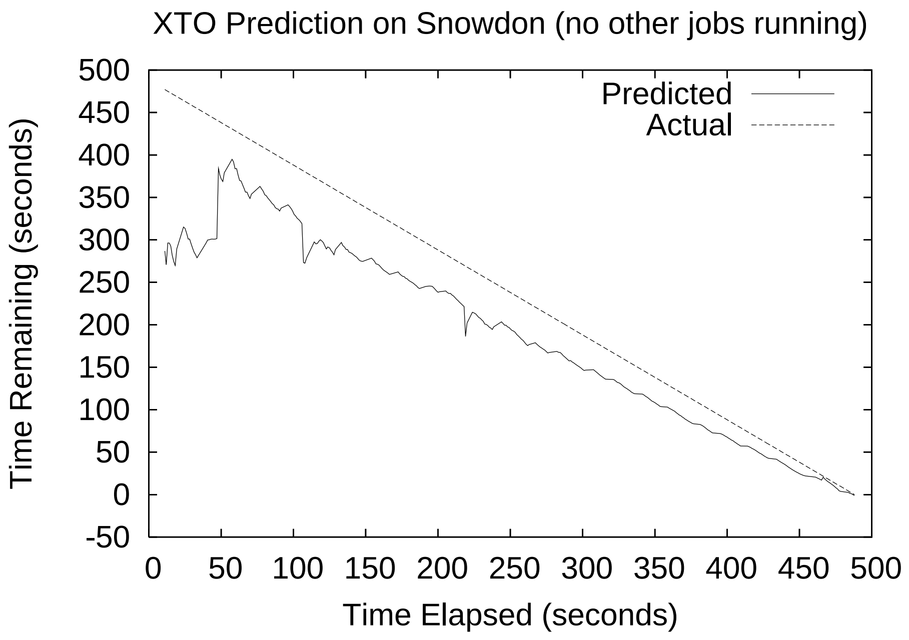

Figure 5.14: Prediction Evaluation on Snowdon

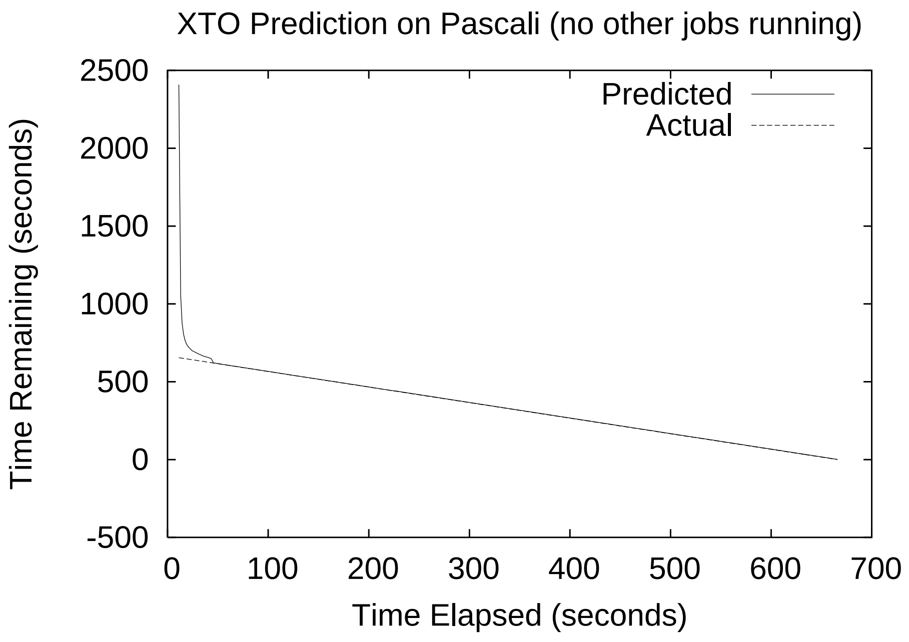

Figure 5.15: Prediction Evaluation on Pascali

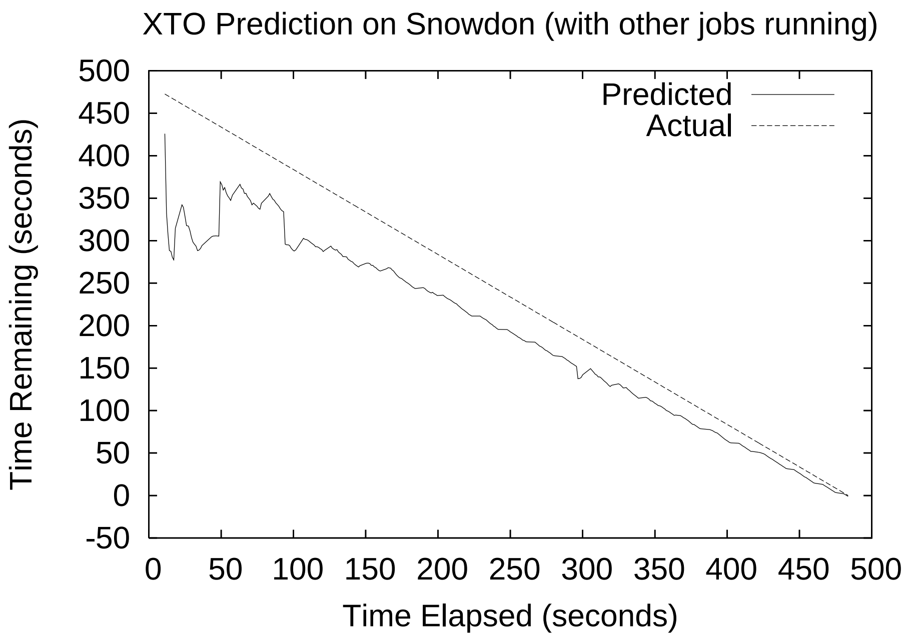

Figure 5.16: XTO Prediction Evaluation including resource performance degradation (Snowdon)

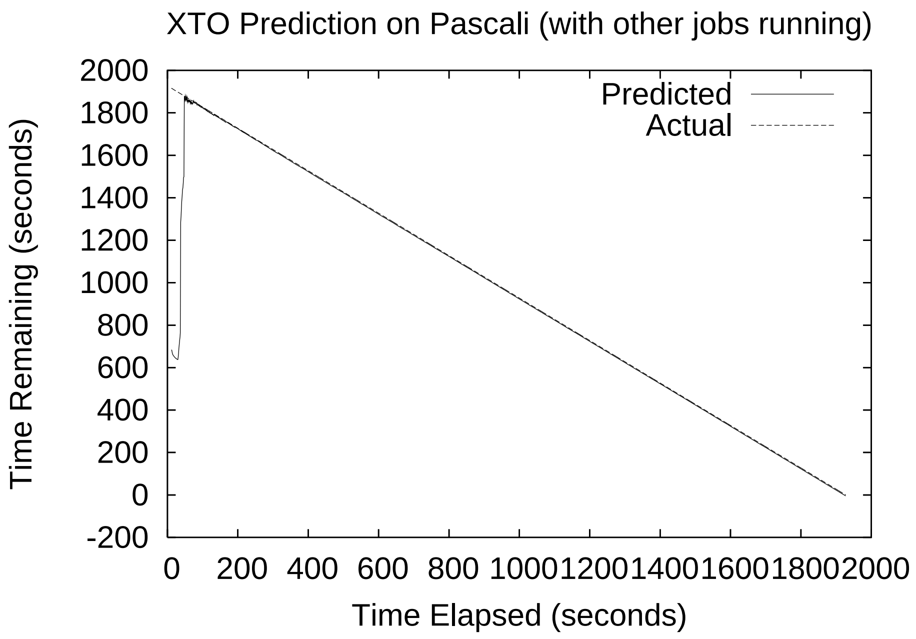

Figure 5.17: XTO Prediction Evaluation including resource performance degradation (Pascali)

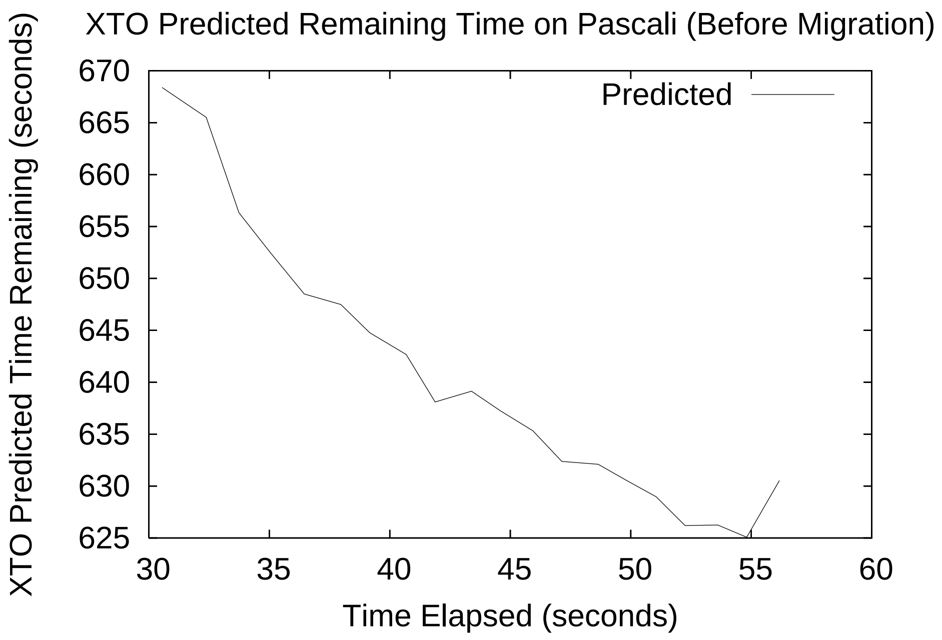

Figure 5.18: XTO Predicted remaining time prior to migration (Pascali)

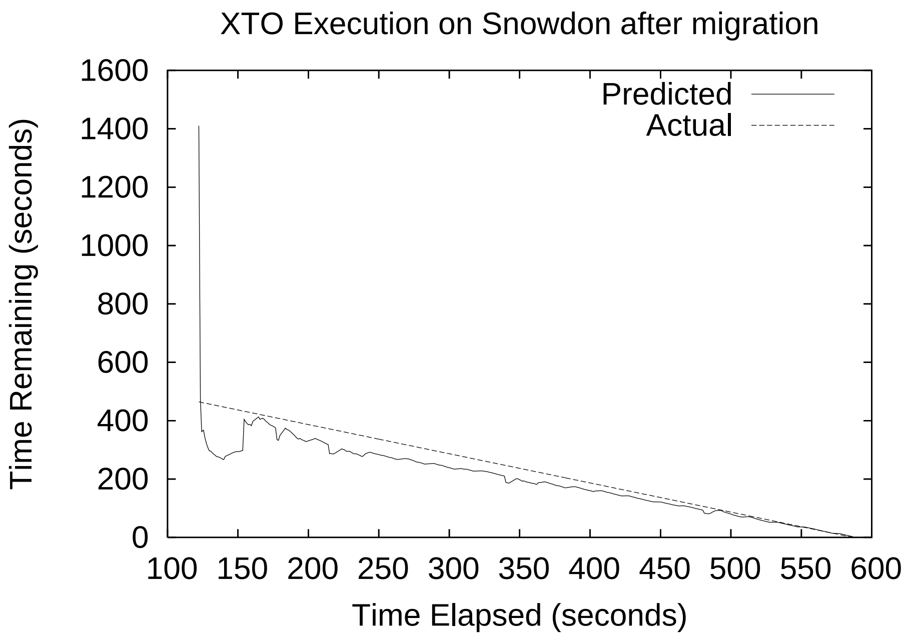

Figure 5.19: XTO Predicted remaining time on Snowdon after migration

5.5.3 Adaptive Service Overhead

There is an overhead associated with the adaptive software, even when job migration is not required. This is due to the fact that the monitoring tool and any calculations used to determine whether migration is required, make use of the same CPU as the application. This suggests that the shorter the periods between each monitoring/decision-making cycle, the greater the overhead. The following experiment is used to evaluate this overhead. Firstly, an application is submitted to a single resource on the WRG and the time it takes to execute recorded. The same application is then run, with the time interval between monitoring/decision-making cycles being varied from 2 seconds through to 120 seconds. The results obtained indicate that the overhead is small even when the interval is only 2 seconds and does not vary significantly with the size of this interval. Specifically, for an application that takes 1319 seconds to complete execution after submission without any monitoring, the time taken with monitoring was in the range. The main cause of this variation was variations in CPU usage rather than overhead due to the adaptive software. This can be seen by normalising the times to the case where the CPU usage is 100%. In this case the minimum time taken is 1108s (time interval between monitoring cycles: 35) and the maximum time taken is 1177s (time interval: 75s). As can be seen in Figure 5.20, there is no indication that the time taken is increasing as the time interval between monitoring cycles is decreased. If the execution times are assumed to be independent of the monitoring period, then the (normalised) times give a sample mean of 1139s with a standard deviation of 23s, which gives a 95% confidence interval of + or – 46s (i.e. + or – 4%).

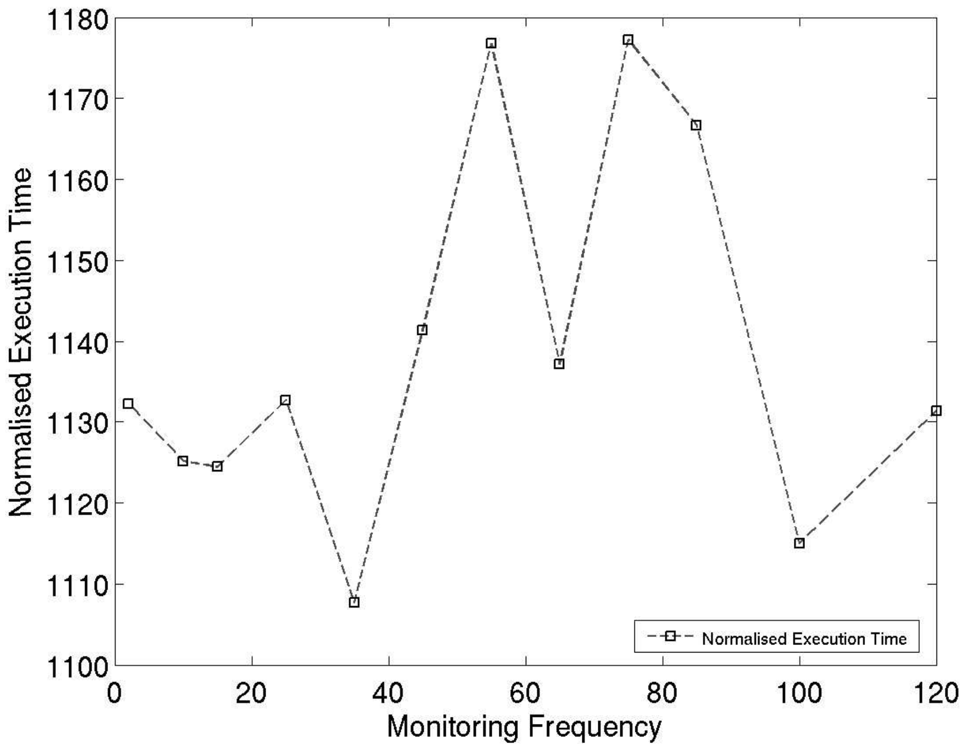

Figure 5.20: Normalised Execution Time

The overhead associated with the adaptive software is so small for the following reasons:

1. The monitoring tool only uses a very small fraction of the CPU.
2. The calculations on which the decision-making as to whether migration is required are simple and therefore do not require much time. Consequently the CPU is only shared between application and decision-making software for very short time intervals.

It is expected that future versions of this software will use more sophisticated decision-making techniques, which are more compute intensive. In this case the overhead may be a more significant issue, particularly when the monitoring/decision-making frequency is high.

5.6 Summary

Our work in this research shows a Grid test-bed and WRG implementation of an adaptive service that supports job migration during run-time to ensure timely job completion. Performance prediction is used to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user specified deadline. A key feature of our approach is that the user is not required to install additional software, or make complex alterations to their code requiring specialist Grid computing knowledge. This is achieved using a reflective technique to bind the adaptive service components to the user’s code. Also, this research proves the adaptive service overhead is very minimal.

Obstacles are those frightful things you see
when you take your eyes off your goal.
— Henry Ford

Chapter 6

Conclusion and Future Work

6.1 Introduction

This chapter summarises the work reported in this thesis by revisiting the research problem and question outlined in chapter one (see section 1.2) so as to reflect on contributions made. Within these discussions the extent to which contributions made address the key research question is reviewed. In addition to this, currently perceived limitations of the contributions made are outlined. Finally, suggestions about possible areas of future research directions are presented.

6.2 Thesis Overview

This thesis has addressed the problem of introducing adaptivity into Grid resource brokering. The adaptive service is a key feature of the architecture presented in Chapter 4, since without it; fulfilment of the user requirements depends entirely on the dynamically varying performance of the resources. A number of issues considered in deciding on an appropriate design of this service were:

1. The adaptive service itself must be implemented in such a way as to ensure that its own presence does not have a negative impact on the application performance (e.g. by significantly increasing the execution time).
2. Overheads associated with job migration (e.g. file transfer times) need to be accounted for when deciding whether migration is an appropriate course of action.

3. In order to make use of the adaptive service the user should not be required to alter their code prior to submission; the system complexities should be transparent to the user.

This research has presented an adaptive service for the Grid, implemented using reflection to ensure transparency to the user and remove any need for the user to alter their application code. The adaptive service has been implemented on the White Rose Grid and shown for two types of applications to provide a significant performance enhancement when resource performance degrades due to an increase in workload. This thesis evaluated the adaptive services which include monitoring, decision making, prediction model, and migration. It has been evaluated through the use of Grid test-bed and the White Rose Grid (WRG) [17]. This evaluation also included measuring the overhead occurs by adaptive service.

This research differ from Gird.It [56], GrADS [2], GridWay [57], ATLAS [55] and AppLeS [50] in the following:

- The goal of adaptation in this research is to optimise the performance of the user's applications in order to meet the requirements. The adaptive service evaluates the resources and if necessary allocates the user's application to better resource.
- The adaptation dispatches with the user's job to the selected resource. Hence, the adaptive service is bound to the user's job and both execute in the same resource. The location of the adaptive service in this research is useful to reduce the network traffic and timely more accurate as the traffic delay might effect the prediction model results in case the adaptive services run remotely. Hence, the adaptive service developed for time constraint jobs.
- When the adaptive service is bound to the user's job and dispatched to the specific resource, adaptation simply uses the prediction model to estimate the remaining execution time. The decision manager depends on the information from the predicted model, historical data and user's requirements. The decision manager decides whether to migrate the job to a new resource or not.

- The adaptive service migrates the job based on the decision manager. The decision takes place during run-time and if the prediction indicates that the application will not finish within the specified time, then the progress of the job is saved, through the use of checkpointing and the resource selector is instructed to find an alternative resource.

6.3 Contributions

The primary contribution is the development of an adaptive framework to ensure the user's requirements are fulfilled during run-time. The framework insulates the user from the complexity of Grid middleware and the Grid resource environment and it consists of two parts:

1. The extension to the simple resource broker includes historical database, which is used by resource selector and estimator components. The resource selector searches the database for historical information about the candidate resources, and the resource with higher probability of failure will be eliminated. The estimator is to predict the queuing time and file transfer time.
2. The adaptive service to ensure the user's requirements has been fulfilled during run-time. The adaptive service's functionality is incorporated into the resource broker and consists of (1) monitoring tool, (2) decision management, and (3) migration engine. The decision manager uses prediction model to estimate expected job completion time and determine whether any observed performance degradation is likely to result in a failure to meet a user specified deadline. The prediction method used here is based on the pattern of CPU usage during run-time, and is therefore expected to display a high level of accuracy only for CPU-intensive applications.

The adaptive service provides an enhancement over the traditional resource broker in terms of the time taken between submission (to the broker) of user requirements and the job finishing execution. Thus the adaptive service is bound to the user application using a reflective technique to insulate the user from the complexity of the Grid environment.

6.4 Limitations and Future Work

The limitation and future work fall into two major parts of the architecture presented in Chapter 4 and they are described as follows.

6.4.1 Adaptive service

A major topic of further investigation is the application run-time prediction. In addition to considering a wider parameter space (i.e. taking account of more factors than just CPU time) it is the intention to develop the prediction to produce a probability distribution as output rather than a single prediction of completion time. The current model does not reflect the statistical nature of the system, whereby the execution time cannot be predicted with any certainty. A probabilistic approach will enable risk of failure to be factored into the decision-making process.

Regarding the decision-making algorithm itself, selecting an appropriate value of *N* (Section 4.8.4) has not yet been investigated in detail. Our intent is to evaluate the effect that altering this parameter has on the prediction accuracy and the migration decision-making for a variety of application execution times and resource usage patterns. This may be supported using simulations. Further, the idea of introducing a “slack” time will be considered, so that migration is initiated if the prediction indicates that the user’s deadline is not going to met with a certain amount of time to spare. Moreover, we plan to investigate the resource selection process, with a view to increasing the performance enhancement that the adaptive service can provide through job migration support.

Checkpointing in this research works with Java applications only. However, checkpointing requires to be extended to allow migration of user’s job written in other languages such as C, C++, etc. The extension should meet the objectives of the adaptive service outlined in Section 1.4, namely that the adaptive service runs in any resource without the need to have special software installed. This will enable the adaptive service to run on any resource in the Grid.

The adaptive resource Broker has been tested in a real Grid environment (see Section 5.5 White Rose Grid experiments), and different types of applications has been submitted to the Broker. However, submitting many applications at once and having more than one user use the Broker has not been tested and is considered to be future work. However, as the instance of adaptive service is bound to the user's application, we believe that the adaptive resource broker can be scalable.

6.4.2 Historical database

The historical database was developed for this research to serve the adaptive service. It is stored in a MySQL database and contains data about the jobs previously run at particular resources. The information such as predicting job failure, transfer time, and queuing time are used to enhance the resources selection process, and thereby increase the likelihood of timely job completion. For example, if a resource often fails, it is unlikely to be selected, compared to a resource that never fails. The resource selector makes use of the historical database by estimating the probability that the job fails to finish within the user specified time constraint.

The historical database developed for this research resides with the adaptive broker and it is a single database consisting of few historical records. However, years of collecting information about the resources will achieve an ideal historical database. Moreover, to reduce access latency, improve data locality, and/or increase robustness, scalability and performance for distributed applications data replication should be considered.

Further research is required into the estimation of overheads associated with job migration, e.g. file transfer, queue waiting times, etc. Our approach is to be based on the use of historical data. Careful consideration of the type of data to be contained in the database and its use in estimating migration times is required. Thus, the historical database can be extended to provide more information on risk of failure. The following is description of what can be implemented:

1. The failure probability of the jobs migrated to specific resource. If the job is going to be migrated, the transfer time and queuing time are overheads. These adversely affect the probability of failure.
2. The failure sensitivity of the jobs in relation to the time of the day. Some resources are likely to fail in finishing the jobs on time specified at specific time of the day.

3. The failure rate in relation to job execution time. Some resources are better for short execution time (e.g. 10 minutes job).

This type of statistic would be important when deciding to migrate the job or even start the job.

He who wonders discovers that this in itself is wonderful.

— M. C. Escher

Chapter 7

References

1. Ferreira, L., *Introduction to Grid Computing with Globus*. 2003: IBM Redbooks.

2. Berman, F., A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, *The GrADS Project: Software support for high-level Grid application development.* International Journal of High Performance Computing Applications, 2001(15 (4)): p. 327–334.

3. Czajkowski, K., S. Fitzgerald, I. Foster, and C. Kesselman. *Grid information services for distributed resource sharing*. 2001. http://www.globus.org/alliance/publications/papers/MDS-HPDC.pdf. access Jan 2006.

4. Berman, F., R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, *Adaptive computing on the Grid using AppLeS.* Parallel and Distributed Systems, IEEE Transactions on, 2003. **14**(4): p. 369.

5. Andrew, S.G., A.W. Wm, and C.T.L. Team, *The Legion vision of a worldwide virtual computer.* Commun. ACM, 1997. **40**(1): p. 39-45.

6. Larry, S., *Toward the 21st century.* Commun. ACM, 1997. **40**(11): p. 28-32.

7. Foster, I. and C. Kesselman, *The grid: blueprint for a new computing infrastructure*. 2nd ed. 2004, San Francisco, Calif.: Morgan Kaufmann. xxvii, 748 p.

8. Leinberger, W. and V. Kumar, *Information power grid: The new frontier in parallel computing?* Concurrency, IEEE [see also IEEE Parallel & Distributed Technology], 1999. **7**(4): p. 75.

9. Mullender, S., *Distributed systems*. 2nd ed. 1993, New York Wokingham: ACM Press; Addison-Wesley. xvi,595p.

10. Tanenbaum, A.S., *Distributed operating systems*. 1995, London: Prentice-Hall International. xvii,582p.

11. Foster, I., C. Kesselman, and S. Tuecke, *The anatomy of the Grid: Enabling scalable virtual organizations.* International Journal of Supercomputer Applications, 2001.

12. Grimshaw, A., A. Ferrari, F. Knabe, and M. Humphrey, *Wide area computing: resource sharing on a large scale.* Computer, 1999. **32**(5): p. 29.

13. Schopf, J., *a general architecture for scheduling on the grid schopf.* special issue of JPDC on Grid Computing, 2002, Argonne National Laboratory.

14. Austin, J., R. Davis, M. Fletcher, T. Jackson, M. Jessop, B. Liang, and A. Pasley, *DAME: searching large data sets within a grid-enabled engineering application.* Proceedings of the IEEE, 2005. **93**(3): p. 496.

15. Othman, A., P. Dew, K. Djemame, and I. Gourlay. *Adaptive grid resource brokering*. Proceedings. IEEE International Conference on
Volume , Issue , 1-4 Dec. 2003 Page(s): 172 - 179. 2003. Hong Kong, China:

16. Blair, G., G. Coulson, and P. Robin. *an Architecture of Next Generation Middleware*. In proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. 1998: Springer-Verlag.

17. Dew, P., J. Schmidt, M.R. Thompson, and P. Morris, *The White Rose Grid: practice and experience.* All hand meeting, 2003.

18. Chiba, S., *Load-Time Structural Reflection in Java*. Proceedings of the 14th European Conference on Object-Oriented Programming Lecture Notes in Computer Science. 2000. pp. 313-336.

19. Pattie, M., *Concepts and experiments in computational reflection*, in *Conference proceedings on Object-oriented programming systems, languages and applications*. 1987, ACM Press: Orlando, Florida, United States.

20. Fabre, J., V. Nicomette, and W. Zhixue, *Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming*, in *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. 1995, IEEE Computer Society.

21. Foster, I. and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit.* The International Journal of Supercomputer Applications and High Performance Computing, 1997. **11**(2): p. 115-128.

22. Frey, J., T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. *Condor-G: a computation management agent for multi-institutional grids*. Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10). 2001.

23. Gentzsch, W. *Sun Grid Engine: towards creating a compute power grid*. Proceedings. First IEEE/ACM International Symposium. pp.35-36. 2001.

24. Jones, J.P. and B. Nitzberg, *Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization*. In Job Scheduling Strategies for Parallel Processing, pp. 1-16, Springer-Verlag, 1999. Lect. Notes Comput. Sci. Vol. 1659.

25. Klaus Krauter, R.B.M.M., *A taxonomy and survey of grid resource management systems for distributed computing.* Software: Practice and Experience, 2002. **32**(2): p. 135-164.

26. Dullmann, D., W. Hoschek, J. Jaen-Martinez, B. Segal, A. Samar, H. Stockinger, and K. Stockinger. *Models for replica synchronisation and consistency in a data grid*. In Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10). 2001.

27. Minoli, D., *A networking approach to grid computing*. ISBN: 0-471-70425-3 2005, Hoboken, N.J.: Wiley. xvii, 377 p.

28. Abramson., D., J. Giddy., and L. Kotler., *High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?* in

Proceedings of the 14th International Symposium on Parallel and Distributed Processing. 2000, IEEE Computer Society.

29. Aagesen, F.A. and C.e.a. Anutariya, *Intelligence in Communication Systems*. IFIP International Conference, INTELLCOMM, ed. L.N.i.C. Science. Vol. 3283. 2004, Bangkok: Springer. 327 p.

30. Hai, Z., *China's e-science knowledge grid environment.* Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications], 2004. **19**(1): p. 13.

31. Humphrey, M. and M.R. Thompson. *Security implications of typical Grid Computing usage scenarios*.10th IEEE International Symposium, pp.95-103. 2001.

32. Foster, I. and C. Kesselman, *The Globus Project: A Status Report*. 1998.

33. Foster, I., C. Kesselman, G. Tsudik, and S. Tuecke, *A security architecture for computational grids,* in *Proceedings of the 5th ACM conference on Computer and communications security*. 1998, ACM Press: San Francisco, California, United States.

34. Wilson, J., *The IETF: laying the Net's asphalt.* Computer, 1998. **31**(8): p. 116.

35. Fitzgerald, S., I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. *A directory service for configuring high-performance distributed computations*. *The Sixth IEEE International Symposium,*pp.365-375, 1997.

36. Allcock, W., J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, *GridFTP: Protocol Extension to FTP forthe Grid.* Grid Forum Internet-Draft, 2001.

37. Service/MDS, I. *http://www.globus.org/toolkit/mds/*. [accessed 2006 16/01].

38. Howes, T.A. and M.C. Smith, *LDAP Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*. Technology Series. 1997: MacMillan Technical Publishing, Indianapolis, Indiana.

39. Microsystems, S., *http://www.sun.com/software/gridware*.[accessed 2006]

40. Baker, R.L., *Making the Net Work: Deploying a Secure Portal on Sun Systems*. Sun Blueprints Ser. Vol. First Edition. 2004: Prentice Hall Professional Technical Reference.

41. Anand, N., A.H. Marty, and S.G. Andrew, *The Legion support for advanced parameter-space studies on a grid.* Future Gener. Comput. Syst., 2002. **18**(8): p. 1033-1052.

42. Michael, J.L., J.F. Adam, A.H. Marty, F.K. John, M.M. Mark, N. Anand, N.-T. Anh, S.W. Glenn, and S.G. Andrew, *Support for extensibility and site autonomy in the Legion grid system object model.* J. Parallel Distrib. Comput., 2003. **63**(5): p. 525-538.

43. Grimshaw, A.S., *Easy-to-use object-oriented parallel processing with Mentat.* Computer, 1993. **26**(5): p. 39.

44. The Globus Resource Specification Langauage, R.v., http://www-fp.globus.org/gram/rsl_spec1.html. [accessed 2006].

45. Foster, I., C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. *A distributed resource management architecture that supports advance reservations and co-allocation*. in Proceedings of the International Workshop on Quality of Service, vol. 13, no. 5, 1999, pp. 27–36.

46. Buyya, R., D. Abramson, and J. Giddy. *Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid.* http://www.csse.monash.edu.au/~davida/papers/hpcasia.pdf. [accessed January 2006].

47. Abramson, D., R. Sosic, J. Giddy, and B. Hall. *Nimrod: a tool for performing parametrised simulations using distributed workstations*. In *Proceedings of the Fourth IEEE International Symposium*, pp.112-121, 2-4 Aug 1995.

48. Grishikashvili, E., D. Reilly, N. Badr, and A. Taleb-Bendiab. *From component-based to service-based distributed applications assembly and management*. *Euromicro Conference, Proceedings. 29th* , pp. 99- 106, 1-6 Sept. 2003.

49. Laddaga, R., P. Robertson, and H. Shrobe, *Introduction to Self-adaptive Software: Applications*. Lecture Notes in Computer Science. **v** (2614). **pp.** 1-5. 2003.

50. Henri, C., O. Graziano, B. Francine, and W. Rich, *The AppLeS parameter sweep template: user-level middleware for the grid*, in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. 2000, IEEE Computer Society: Dallas, Texas, United States.

51. Holly, D., C. Henri, and B. Fran, *A decoupled scheduling approach for the GrADS program development environment*, in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. 2002, IEEE Computer Society Press: Baltimore, Maryland.

52. Kennedy, K., M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, R. Aydt, and D. Reed. *Toward a framework for preparing and executing adaptive grid programs*. International Parallel and Distributed Processing Symposium 2002. http://www.globus.org/alliance/publications/papers/ngs-ipdps02.pdf. [accessed January 2006].

53. Wolski, R., N. Spring, and J. Hayes, *The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing.* Future Generation Computing Systems, 1999. **15**(5-6): p. 757-768.

54. Ribler, R.L., J.S. Vetter, H. Simitci, and D.A. Reed. *Autopilot: adaptive control of distributed applications*. HPDC, *The Seventh International Symposium,* pp.172-179, 28-31 Jul 1998.

55. Baldeschwieler, J.E., R. Blumofe, and E. Brewer, *ATLAS: an infrastructure for global computing*, in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*. 1996, ACM Press: Connemara, Ireland.

56. Aldinucci, M., S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo, *Components for high performance grid programming in the grid.it project*. in Proceeding of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo France). Springer, January 2005.

57. Huedo, E., R. Montero, and I.M. Llorente, *An Experimental Framework for Executing Applications in Dynamic Grid Environments.* Intl. J. Software Practice&Experience., ICASE Technical Report. 2002.

58. Montero, R.n.S., E. Huedo, and I.M. Llorente, *Grid Resource Selection for Opportunistic Job Migration*. Lecture Notes in Computer Science. Volume 2790, Pages 366 - 373. 2003.

59. Padgett, J., K. Djemame, and P. Dew, *Predictive adaptation for service level agreements on the Grid.* International Journal of Simulation: Systems, Science & Technology, 2006. **vol 7**: p. 29-42.

60. Padgett, J., K. Djemame, and P. Dew. *Predictive run-time adaptation for service level agreements on the Grid*. in *Proceedings of the 21st UK Performance Engineering Workshop (UKPEW'2005)*. 2005. University of Newcastle upon Tyne.

61. Lamehamedi, H., B. Szymanski, Z. Shentu and E. Deelman. *Data replication strategies in grid environments.* in Algorithms and Architectures for Parallel Processing IEEE. 2002.

62. Tanenbaum, A. and M. Steen, *Distributed Systems: Principles and Paradigms*. 1st ed, ed. Prentice-Hall. 2002, Englewood Cliffs, NJ, U.S.A.

63. Ratner, D., *Roam: A Scalable Replication System for Mobile and Distributed Computing* 1998, PhD thesis. University of California: Los angeles.

64. Chu, W., *Optimal File Allocation in a Multiple Computer System.* Computers, IEEE Transactions, 1969. **C-18**: p. pp 885-889.

65. Dowdy, L. and D. Foster, *Comparative models of the file assignment problem.* ACM Computing Surveys, 1982. **14**(2): p. 287-313.

66. wolfson, O. and S. Jajodia, *An algorithm for dynamic data distribution.* 2nd Workshop on the Managment of Replicated Data, 1992: p. 62-65.

67. Acharya, S. and S. Zodonik, *An efficient scheme for dynamic data replication.* technical report CS-93-43, Brown University, 1993.

68. Tierney, B., R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany, *A Grid Monitoring Service Architecture.* The Global Grid Forum, 2002. GWD-GP-16-2. January 2002.

69. Cooke, A., A.J.G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. Callaghan, *R-GMA: An Information Integration System for Grid Monitoring*. Lecture Notes in Computer Science, Volume 2888, Pages 462 - 481 Oct 2003.

70. Wolski, R., *Dynamically forecasting network performance using the Network Weather Service.* Cluster Computing, 1998. **1**(1): p. 119.

71. Gunter, D. and B. Tierney. *NetLogger: a toolkit for distributed system performance tuning and debugging*. In Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management, March 2003.

72. GGF. *Global Grid Forum*. [accessed January 2006].

73. Stallings, W., *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2,* ed. 3rd. 1998: Wiley Publishing House.

74. Massie, M., B. Chun, and D. Culler, *The Ganglia distributed monitoring system:Design, implementation, and experience.* Parallel Computing, 2004(30): p. 817-840.

75. Barak, A., O. La-adan, and A. Shiloh. *Scalable cluster computing with MOSIX for LINUX*. in *Linux Expo '99*. 1999.

76. Basney, J., M. Livny, and T. Tannenbaum, *Throughput Computing with Condor.* HPCU news, 1997. **1**(2).

77. Kohl, J., P. Papadopoulos, and G. Geist. *Cumulvs: Collaborative infrastructure for developing distributed simulations*. in *8th SIAM Conference on Parallel Processing for Scientic Computing*. 1997. Minneapolis, MN, USA.

78. LI, J. and J. TSAY. *Checkpointing Message-Passing Interface (MPI) Parallel Programs*. in *Pacific Rim International Symposium on Fault Tolerant Systems*. pp.147-152, 15-16 Dec 1997.

79. Silva, L. and J. Silva. *System-level versus User-defined Checkpointing*. in *Seventeenth Symposium on Reliable Distributed Systems*. In The 17th IEEE Symposium on Reliable Distributed Systems. **pp.** 68.1998. West Lafayette.

80. Smith, M.C., *Procedural Reflection in Programming Lanuages*, in *MIT Laboratory of Computer Science Technical*. 1982, MIT: Cambrige, Mass.

81. Stroud, R. and Z. Wu, *Using Metaobject Protocols to Satisfy Non-functional Requirements*, in *Advances in Object-Oriented Metalevel Architectures and Reflection*. 1996, CRC press. p. 21-52.

82. Maes, P., *Concepts and experiments in computational reflection*, in *Conference proceedings on Object-oriented programming systems, languages and applications*. 1987, ACM Press: Orlando, Florida, United States.

83. Takuo, W. and Y. Akinori, *Reflection in an object-oriented concurrent language*, in *Conference proceedings on Object-oriented programming systems, languages and applications*. ACM Press: San Diego, California, United States, 1988.

84. Kiczales, G., J. des Rivieres, and D. Bobrow, *The Art of Metaobject Protocol*. 1991: ISBN:0-262-61074-4. MIT press.

85. Tatsubori, M. and S. Chiba. *Programming Support of Design Patterns with Compile-time Reflection*. in *Proceeding of Workshop on reflective Programming in C++ and Java*. 1998. Center for Computational Physics, University of Tsukuba, Japan.

86. Hanspeter, M. and S. Christoph, *The Oberon-2 Reflection Model and Its Applications*, in *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*. 1999, Springer-Verlag.

87. Ian, W. and J.S. Robert, *From Dalang to Kava - The Evolution of a Reflective Java Extension*, in *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*. 1999, Springer-Verlag.

88. Yasuhiko, Y., *The Apertos reflective operating system: the concept and its implementation*, in *conference proceedings on Object-oriented programming systems, languages, and applications*. 1992, ACM Press: Vancouver, British Columbia, Canada.

89. Ashish, S., S. Aamod, and H.C. Roy, *Reflective ORBs: Supporting Robust, Time-Critical Distribution*, in *Proceedings of the Workshops on Object-Oriented Technology*. 1998, Springer-Verlag.

90. Stankovic, J.A. and S.H. Son. *Architecture and object model for distributed object-oriented real-time databases*. *First International Symposium*, pp.414-424, 20-22 Apr 1998.

91. John, A.S., R. Krithi, N. Douglas, H. Marty, and W. Gary, *The Spring System: Integrated Support for Complex Real-Time Systems*. Real-Time Systems, Volume 16, Issue 2 - 3, May 1999, University of Virginia.

92. Bondavalli, A., J. Stankovic, and L. Strigini. *Adaptable Fault Tolerance For Real-Time Systems*. In Responsive Computer Systems: Toward Integration of Fault-tolerance and Real-time, D. F. Ed. (Eds.), Kluwer Academic Publishers, 1995, pp.187-208 1994, University of Massachusetts.

93. Tushar Deepak, C. and T. Sam, *Unreliable failure detectors for reliable distributed systems.* J. ACM, 1996. **43**(2): p. 225-267.

94. Agha, G. and D. Sturman, *A Methodology for Adapting Patterns of Faults.* Foundations of Dependable Computing: Models and Frameworks for Depnedable Systems, 1994. **1**: p. 23-60.

95. Richard, H., H. Andrew, and D. Douglas, *FlexiNet\—a flexible component oriented middleware system*, in *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. 1998, ACM Press: Sintra, Portugal.

96. Thomas, L., *OpenCorba: A Reflektive Open Broker*, in *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*. 1999, Springer-Verlag.

97. A. Nguyen-Tuong and A. S. Grimshaw. *Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications*. Parallel Processing Letters, 9(2):291–301, 1998, University of Virginia.

98. Shigeru, C., *A metaobject protocol for C++*, in *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. 1995, ACM Press: Austin, Texas, United States.

99. Pawlak, R. and L. Seinturier, *Implementation of an Event-Based RT-MOP*, in *Research Report CNAM-CEDRIC 98-04*. 1998.

100. Buyya, R., D. Abramson, and J. Giddy. *A case for economy grid architecture for service oriented grid computing*. In p*roceedings 15th International*, pp.776-790, Apr 2001.

101. Henri, C. and D. Jack, *NetSolve: a network server for solving computational science problems*, in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Volume 00*. 1996, IEEE Computer Society: Pittsburgh, Pennsylvania, United States.

102. Kofler, M., *MySQL*. Books for professionals by professionals. 2001, Berkeley, CA: Apress,. xi, 659 p.

103. Gosling, J., *The Java language specification*. 2nd ed. ed. 2000, Boston ; London: Addison-Wesley. xxv,505p.

104. Warren, S., E.T. Valerie, and T.F. Ian, *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance*, in *Proceedings of the Job Scheduling Strategies for Parallel Processing*. 1999, Springer-Verlag.

105. MacLaren, J. *Advance reservations state of the art*. Technical Report, Global Grid Forum, http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html [accessed 2006 20 Feb].

106. Jackson, D., Q. Snell, and M. Clement, *Core Algorithms of the Maui Scheduler*. Lecture Notes in Computer Science. Volume 2221, pp. 87. 2001.

107. The public Netperf homepage. *http://www.netperf.org/netperf/NetperfPage.html*. [accessed Feb 2006.]

108. Thrulay. *www.internet2.edu/~shalunov/thrulay/*. [accessed 2006 20 Feb].

109. Kurose, J.F. and K.W. Ross, *Computer networking: a top-down approach featuring the Internet*. 3rd ed., Boston; London: Pearson/Addison Wesley. xxiv, pp.821. 2005.

110. Brian, R. and K.G. David, *Static dependent costs for estimating execution time*, in *Proceedings of the 1994 ACM conference on LISP and functional programming*. 1994, ACM Press: Orlando, Florida, United States.

111. Freund, R.F., *Optimal selection theory for superconcurrency,* in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. 1989, ACM Press: Reno, Nevada, United States.

112. Iverson, M.A., F. Ozguner, and G.J. Follen. *Run-time statistical estimation of task execution times for heterogeneous distributed computing*. In *Proceedings of the High Performance Distributed Computing (HPDC '96)*, IEEE Computer Society, Washington, DC, 263 August 06 - 09 1996.

113. Grinstein, S., J. Huth, and J. Schopf. *Resource Predictors in HEP Applications*. in *Computing in High Energy Physics*. Sep 2004. Interlaken, Switzerland.

114. Smith, W., I. Foster, and V. Taylor, *Predicting Application Run Times Using Historical Information*. Lecture Notes in Computer Science. 1998. 122.

115. Gregor von Laszewski, I.F.J.G.P.L., *A Java commodity grid kit.* Concurrency and Computation: Practice and Experience, 2001. **13**(8-9): p. 645-662.

116. Wrightson, K. and J. Merlino, *Mastering Unix*., San Francisco; London: Sybex. xlii, pp. 897. 2001.

117. Barak, A., O. Laadan, and A. Shiloh. *Scalable cluster computing with MOSIX for LINUX*. in *Linux Expo 99*. 1999.

118. Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano, *OpenJava: A Class-Based Macro System for Java*. Lecture Notes in Computer Science. pp. 117. 2000.

119. The London e-science. *http://www.lesc.imperial.ac.uk/projects/SGE-GT4.html*. [accessed march 2006].

120. Nadeem, S., P. Dew, and K. Djemame, *XTO Grid Services on the White Rose Grid: Experiences in building an OGSA Grid Application,*, in *DAME Technical Report,*. 2003, University of Leeds.