Automated Grid Fault Detection and Repair

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Leslie Luyt

Grahamstown, South Africa
2012

Abstract

With the rise in interest in the field of grid and cloud computing, it is becoming increasingly necessary for the grid to be easily maintainable. This maintenance of the grid and grid services can be made easier by using an automated system to monitor and repair the grid as necessary. We propose a novel system to perform automated monitoring and repair of grid systems. To the best of our knowledge, no such systems exist. The results show that certain faults can be easily detected and repaired.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2012) [1]:
D.1.3[Programming Techniques] Concurrent Programming — Distributed Programming
K.4.3[Organizational Impacts] Automation
K.6.4[System Management]
General-Terms: Reliability, Security, Management

Acknowledgements

The production of this thesis has taken many months and it would be unfair not to acknowledge all those who have assisted me. I would therefore like to thank all those who have contributed and assisted, in any way, to the completion of this project. There are a number of people I would like to specially note.

Firstly, I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Eastel, Bright Ideas Project 39, and THRIP through the Distributed Multimedia Centre of Excellence in the Department of Computer Science at Rhodes University.

Secondly, I would like to acknowledge my parents for all the support, guidance, funding and love they have given me over the years. I know it would not have been possible for me to have had this great opportunity had it not been for them.

Thirdly, I would like to extend my thanks to Jonas Bardino of the Minimum Intrusion Grid project whose input was invaluable and without whose assistance this thesis would not have been nearly as productive as it has been.

Finally, I would like to thank and acknowledge my supervisor, Dr. Karen Bradshaw. Without her much valued input, support and guidance this thesis would not have been possible.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Grid computing has evolved considerably since its first inception in 1994 [18]. With the shift in computing towards multi-core processors and services from the cloud, grid computing has become more important than ever and has started to permeate daily computer usage. As such, it is becoming increasingly more important to understand and be able to easily manage these large grid systems. To manage large grid systems a monitoring tool needs to be used. Current monitoring tools, such as Big Brother [24] and Nagios [32], allow for the monitoring of physical system attributes with little to no regard to the software systems running on the underlying hardware. This type of monitoring is perfectly acceptable for a machine that provides a number of generic services, however it is not ideal for grid servers as there are numerous problems that cannot be detected purely by monitoring the executing machine's hardware.

Grid systems are defined as hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [13]. Thus, since it is possible to reliably monitor the hardware component of the grid system, monitoring the software component of the grid system becomes of critical importance in keeping the grid system online and operational.

1.1 Problem Statement

The shift towards grid computing has been exceptionally fast with many services now being served from the cloud instead of from a fixed server. However, no automated maintenance management tool for grid systems currently exists, which makes grid maintenance difficult

and does not allow for observation of trends on grid systems. Given that maintenance of the grid system is a critical part of keeping a grid computing service running, it is vital that system administrators have a way of ensuring that the system is operating correctly. This research is novel, in that there has been little or no previous work done in this field with this end goal in mind and at the time of writing, to the best of our knowledge, no automated maintenance systems existed.

1.2 Research Methodology

The research involves creating a prototype automated grid maintenance system as well as a method whereby this system can identify the occurrence of faults on the grid system. Thereafter, testing is conducted to ascertain to what extent the system can successfully detect and repair any errors that occur on the grid system. Although the proposed system should be designed such that it can operate on multiple types of grids and use a variety of grid products, testing of the system using multiple grid types and systems is beyond the scope of the research.

1.3 Project Outcomes

1.3.1 Expected End-product

The minimum expected end-product of the project is an extensible grid management and repair system. This grid repair system should include the ability to observe trends on the grid, as well as manage grid settings to improve efficiency. Furthermore, a trend identification system, placed within an artificial intelligence system, allows for real-time automatic optimisation and repair of the grid, thus allowing the grid to work consistently at peak efficiency.

1.3.2 End-product Requirements

The grid repair system should be modularly designed, with each module providing an overview of what is happening on the grid with respect to the grid metric it was designed to monitor.

The grid repair system should make it easy to identify trends and thus, usage tests of how well the interface displays trends should be conducted. It is expected that given certain metrics, it may be difficult to identify trends against background noise and thus some metric modules may fail some of the devised tests.

Given that trends can be identified for a specified metric module, results can be extracted in terms of how well the system can identify a known simulated trend and when the grid is not running optimally.

It should be possible to extract generic results and statistics from the grid for use by the grid administrator, thereby providing the system administrator with basic and advanced statistics about what is going on within the grid server and network, such as the load on a single server.

1.3.3 End-product Evaluation

The end-product system should be evaluated with respect to all the requirements described above to confirm that the grid management and repair system can effectively help grid administrators secure, manage, and adapt their grid to suit its usage and needs.

1.4 Thesis Organisation

This thesis includes the following chapters. The first chapter is intended as an introduction to the work, presenting background information and the problem to be solved.

Chapter 2 surveys the body of work relevant to this study.

Chapter 3 presents background information on grid computing and the grid systems currently in use.

Chapter 4 presents the current state of management used in grid computing.

Chapter 5 presents and describes the design and research ideals used in the production of this thesis.

Chapter 6 focuses on the implementation and any important factors that arise from this or differ from the research design ideals described in the previous chapter.

Chapter 7 presents and discusses the results obtained from the RAM metric.

Chapter 8 presents and discusses the results obtained from the grid jobs metric.

Chapter 9 critically analyses and discusses the results presented in the previous two chapters.

Chapter 10 compares the system introduced in this thesis to other existing systems that are designed to perform a similar function, albeit not specialised for grid systems.

Chapter 11 summarises the contributions of this research while presenting and discussing the final conclusions.

Chapter 2

Background Knowledge

2.1 Introduction

This chapter introduces and defines the main concepts and issues related to creating an extensible grid management and repair system to facilitate grid management and trend identification.

We outline the advantages and disadvantages of the programming languages considered for the implementation of the grid repair system. In addition, intelligent learning using neural and Bayesian networks, and expert computing systems are introduced for the purpose of trend identification and system repair.

2.2 Programming Language Selection

2.2.1 PHP

2.2.1.1 Description of PHP

PHP is a widely used, general-purpose, object-oriented, HTML-embedded, open-source, server-side, high-level scripting language commonly used to produce dynamic web pages. PHP is commonly used on Linux Apache web servers, however, it can be deployed on most web servers and operating systems. There exists a wide range of PHP frameworks that

offer features similar to other web application frameworks, such as CakePHP[1], Symfony[2], CodeIgniter[3], and Zend Framework[4]. PHP also includes a large range of extensions included in the PEAR module that simplify many advanced tasks. [36]

2.2.1.2 Benefits

PHP is extremely widely used and holds a huge installed user base with many code examples and discussions available. PHP also has a complete object-oriented structure which includes: public, private, abstract, final, and interface keywords and concepts. It also contains most common programming structures, such as switch statements, do-while loops, and decrement and increment operators.

PHP is commonly said to be faster and more efficient for complex programming tasks and trying out new ideas, and is considered to be more stable and less resource-intensive than many other scripting languages. PHP also has a less-confusing and stricter format without losing flexibility and is much easier to integrate into existing HTML. [36]

2.2.1.3 Limitations

PHP has a poor security history with many remotely exploitable vulnerabilities appearing if best practice programming rules are not followed. Hosting PHP applications on a server requires careful and constant attention to deal with these security risks; however, there are advanced protection patches especially designed for production server environments.

PHP also has a weak type system with all variables defaulting to NULL. This weak type system makes it difficult to guarantee exactly what type of data is being dealt with and may cause a piece of data to be used incorrectly. It also provides very poor and inefficient base-types, with the array type doubling as a list and inefficient dictionary data structure.

There is no namespaces concept in PHP and the built-in library has a wide range of naming conventions, forcing some functions to have prefixes to denote their source. However, this limitation has mostly been mitigated by placing functions into classes and is soon to be fixed by the introduction of namespaces in the PHP 5.3 development trunk. [36]

[1] http://cakephp.org/
[2] http://www.symfony-project.org/
[3] http://codeigniter.com/
[4] http://framework.zend.com/

2.2.2 Python

2.2.2.1 Description of Python

Python is an interpreted, interactive, cross-platform, dynamic object-oriented, and very-high-level programming language (VHLL) that provides strong integration and support when combined with other languages. Python is simpler, faster to process, and more regular than classic high-level languages [38]. Python also comes with an extensive set of standard libraries, providing basic functionality in many areas by simply using and including the correct library. Python is stated as boasting a multitude of features, such as: exception-based error handling; high portability; full modularity; supporting hierarchical packages; intuitive object orientation; very clear, readable syntax; and very high level dynamic data types. [26, 41]

2.2.2.2 Benefits

Python has been designed to be modular, with a small kernel that is extended by importing extension modules, or packages. The standard Python distribution includes a diverse library of extensions for operations ranging from string manipulations and regular expression handling, to Graphical User Interface generators, operating system services, and debugging and profiling tools. New Python extension modules can be created to extend the language, and can be written in Python, C, or C++. [26, 39, 41]

Python, as with most other scripting languages, supports numerous high-level constructs and data structures that enable programmers to write shorter programs than those, with similar functionality, written in C, C++, C#, or Java. Where Java requires a loop, Python may require only a single line statement to perform the same operation. This significantly increases the maintainability and readability of code, thereby producing a better end-product than is possible with more traditional languages. [21, 39, 41]

2.2.2.3 Limitations

Since Python is an interpreted language, some performance loss is to be expected. This is a direct result of hardware-independent byte-code being converted to hardware-dependant byte-code, by being run through an interpreter before execution can occur. However, Python code runs with reasonable performance, sufficient not to be a significant drawback;

and in most cases will only run marginally slower than C code for the same tasks. The Python coding style used can also alleviate this problem by using extensions, such as the Numeric extension, which provides good performance when working with large arrays of numbers. [21, 26, 39, 41]

2.3 Web Technology Evaluation

To build a web management portal that provides feedback to system administrators, it is necessary to evaluate and understand the available technologies for hosting the end-product as this may affect what is and is not possible in the web application itself. This section evaluates the common tools for hosting of web applications and focuses on solutions using Apache since it is supported by most operating systems and is the most popular and widely used web server.

2.3.1 mod_python

Mod_python is an Apache extension module that embeds the Python interpreter within the Apache web server processes. This means that Apache does not have to start a new Python interpreter for each request and thus allows Apache to execute Python code much faster than traditional CGI scripts. This embedded approach also allows access to advanced features such as persistent database connections and access to Apache internals. It also provides a set of standard handlers for alternative development frameworks, as well as a set of utility objects and functions for common web development tasks such as cookie processing and session management. Mod_python is often likened to mod_php by PHP developers, however this is not the case as mod_python provides the full Python back-end and is not "Python intermixed with HTML". However mod_python can emulate CGI and work in a "Python Server Pages" mode similar to Java Server Pages, which is "HTML intermingled with Python". [29, 40]

Applications developed using mod_python are Apache-specific and not easily portable to other web server platforms [29]. However since Apache is already supported by a large number of operating systems and is by far the most popular web server in the world, this is not a huge problem. Shortly before the release of version three, mod_python was adopted by the Apache Software Foundation and became an official subproject of the

Apache HTTP Server project; however, there have been no significant updates to the module since early 2007. [11, 30, 45]

One of the downfalls of mod_python is that each Apache child process must load the whole Python interpreter, even if it does not use it, thus making the web server run much slower [11]. Mod_python is also linked to a specific version of libpython, and thus it is not possible to switch from an older version of Python to a newer without recompiling mod_python [40].

2.3.2 mod_wsgi

The Web Server Gateway Interface, or WSGI for short, is a unification of the application programming interface. [40] This means that a WSGI application can be deployed on any gateway interface that supports WSGI wrappers. The Python standard library contains its own WSGI server, wsgiref, which is a small web server that can be used for testing. Since WSGI applications are backwards compatible with standard cgi, a WSGI application can run on any Apache server with mod_python, FastCGI or mod_wsgi installed [11, 31].

Mod_wsgi is a simple to use Apache extension module that can host any Python application that supports the Python WSGI interface. Mod_wsgi has two primary modes of operation: embedded mode and daemon mode. Embedded mode works in a similar way to mod_python, executing the Python application within the context of the normal Apache child processes. Daemon mode, on the other hand, works similarly to FASTCGI solutions, spawning a separate process to handle each Python application. For security and stability reasons it is suggested to use daemon mode rather than the embedded mode, as the latter may allow an attacker to gain access to Apache child processes. Best performance can be achieved using the embedded mode, but this requires careful configuration of the Apache MPM settings away from the default of serving static media and PHP hosted applications. [11, 31, 40]

The mod_wsgi module is written in C code directly against the internal Apache and Python application programming interfaces. This means that it has a lower memory overhead and performs better than existing WSGI adapters for mod_python or alternative FASTCGI/SCGI/CGI solutions when hosting WSGI applications. [11, 31]

2.3.3 Web Programming Frameworks

A Web framework is a collection of packages or modules that allow developers to write Web applications or services without having to handle such low-level details as protocols, sockets or process/thread management [40]. High-level Web frameworks commonly provide combinations of the following components to the end developer: a base HTTP application server, a storage mechanism such as a database, a template engine, a request dispatcher, an authentication module, and an AJAX toolkit. The majority of Web frameworks work exclusively on the server-side, however with the increased use of client-side programming, using AJAX and Javascript, they are starting to provide modules and mechanisms to assist developers with these tasks.

Web frameworks commonly require the developer to adhere to the programming conventions required by the framework for seamless creation of applications where the responsibility for communication and infrastructure are delegated to the framework. Some common available Web frameworks are: Django[5], PylonsHQ[6], Zend Framework[7], and Zope[8].

2.4 Artificial Intelligence

2.4.1 Neural Networks

2.4.1.1 Neural Network Description

An artificial neural network is a computational model defined by a directed graph, where each node, or vertex, in the graph has inputs and outputs. These inputs and outputs form the edges of the graph and are associated with a weight function, or a value. This network provides a practical method for learning real, discrete, and vector valued functions from examples. After the learning phase has been completed the associated weights will provide the required output for all training data. However, these weights may be difficult to explain and interpret, and all that can be inferred is just a numerical relationship between two nodes. [6, 28]

[5] `https://www.djangoproject.com/`
[6] `http://pylonshq.com/`
[7] `http://framework.zend.com/`
[8] `http://www.zope.org/`

2.4.1.2 Neural Learning

The artificial neural network learns by taking in a list of example inputs and the corresponding correct output for each input. For every input given, it uses the current network to compute an output. It then calculates the error between the given output and the current output, and adjusts the weights by backward propagation to minimise the error. After the weights have been adjusted it re-computes the input example. This process iterates until the network correctly produces the required output for each input, or the learning process returns with an error stating that the current network configuration cannot correctly simulate the function that maps the given input to the given output. The system then needs to alter the structure of the network, by adding or deleting more nodes or layers as appropriate, and re-start the learning process.

Once the learning process has been completed successfully, we have a correct neural network structure and weights for the function we wish to emulate. We can then proceed to feed in inputs for which we do not know the output and observe what the neural network produces. The final output produced by the neural network may not necessarily be correct, as there may be missing cases in the example input and output, which lead to only a partial solution being formed. [6, 28]

2.4.2 Bayesian Networks

2.4.2.1 Bayesian Network Description

A Bayesian network is a probabilistic graphic model defined by a directed acyclic graph; where each node represents a random variable and the edges between nodes represent probabilistic dependencies between the random variables associated with the connected nodes. After the learning phase has been completed the probabilities associated with the edges will provide the required solution. These probabilities tend to be relatively easier to interpret than the neural network weights. [5, 6, 28]

2.4.2.2 Bayesian Learning

When the structure of a Bayesian network is unknown at the outset, it is necessary that we learn the structure from the given training data. This problem is known as the BN

learning problem, which can be stated informally as follows. Given training data and prior information, estimate the network structure and the parameters of the joint probability distribution in the BN. Once this initial estimation has been done, the system falls into one of the four cases shown in Table 1.

Table 1: Four cases of BN learning problems

Case	BN structure	Observability	Proposed learning method
1	Known	Full	Maximum-likelihood estimation
2	Known	Partial	EM (or gradient ascent) MCMC
3	Unknown	Full	Search through model space
4	Unknown	Partial	EM + search through model space

For case one, the training can be performed by maximising the log-likelihood of each node independently.
For case two, the expectation maximisation algorithm can be used to find a locally optimal maximum-likelihood estimate of the parameters.
For case three, the goal is to find a directed acyclic graph that explains the data. An approach to solving this NP-hard problem is to take the naïve approach and assume that each node is conditionally independent, thereby simplifying the problem.
For the fourth case, finding the directed acyclic graph is an intractable problem. Thus, the practical approach is to use asymptotic approximation to the posterior called the Bayesian information criterion. This approach marginalises out the hidden nodes and parameters. [5, 6, 28]

2.4.3 Expert Systems

An expert system is an advanced decision making system that attempts to emulate the decision making ability of a human expert by reasoning about the known knowledge [20]. Expert systems use a knowledge base of known knowledge conditions and an inference engine that uses the knowledge base to generate additional knowledge from the known input facts. The knowledge base is essentially a set of rules that are constructed in an "if .. then" format that allow additional knowledge to be understood if the conditions of the if statement are satisfied. The most prominent and well known tool for creating expert systems is Prolog, which was created by Alain Colmerauer and was one of the first logic programming languages available [23].

A Python based expert system tool is Pyke which integrates logic programming into Python and provides an inference engine that can operate on both forward-chaining (data

driven) and backward-chaining (goal oriented) rules [37]. To perform forward-chaining Pyke attempts to match the "if" clause of a rule that matches the already known facts. When a match is found, the facts in the rule's "then" clause are added to the list of known rules and this may fire additional rules if the "then" clause matches another rule's "if" clause, thereby causing a chaining effect. This forward-chaining procedure continues until no further rules are found that match with the list of known facts, and there is no limit place on the recursion performed by the forward-chaining procedure. Since the forward-chaining rule's "if" pattern contains many statements where each statement could be another rule itself, Pyke uses "foreach-assert" syntax for forward-chaining rules. To perform backward-chaining Pyke searches for a rule whose "then" statement matches the goal requested. Once such a rule is found, Pyke then attempts to recursively prove all of the sub-goals listed in the "if" clause of the rule. If all of the sub-goals can be proven, then the original rule succeeds and the requested goal is proven. Since backward-chaining searches the "then" section before checking the "if" section, the rules are "then-if" rules. However, "then-if" rules do not make much sense and thus Pyke code uses "use-when" syntax for backward-chaining rules.

Pyke can be embedded into any Python program and can automatically generate Python programs by assembling individual functions into complete call graphs while ensuring that each function's requirements are completely satisfied before any function is executed [37]. Pyke also supports multiple fact bases and multiple rule bases that allow modularly created programs to make use of the inference engine and compute rule and fact bases from each module at the same time.

Pyke also includes a feature called backtracking, which assists backward-chaining rules and gives them much more power. Backtracking allows Pyke to move back up the execution tree when a rule fails, and then another solution can be attempted for the previous rule [37]. If another solution can be found for this rule, Pyke continues the execution using this new solution. Otherwise, Pyke continues back up the execution tree until it either finds an alternate solution for one of the rules or returns to the base rule, at which point it declares the base rule as having failed. This type of execution model is not available in traditional programming languages, such as Python, and coupled with Pyke's inherent recursive programming gives the Pyke expert system far more power than any sequential programming language can achieve.

Chapter 3

Grids

3.1 Grid Software

3.1.1 Introduction

The idea of the Grid began in 1994 when Rick Stevens and Tom DeFanti proposed establishing temporary links among eleven high-speed research networks to create a national research network. Following this proposal a small team, guided by Ian Foster, successfully created a set of new protocols that allowed applications to be run on computers across the country. The success of this experiment caught the attention of DARPA (the Defence Advanced Research Projects Agency) and their funding led to the first release of the Globus Toolkit in 1997 [18]. Globus was soon a huge success and was deployed on eighty sites around the world, clearly showing high interest in this field of research. The Globus success led to both NASA and the United States Department of Energy each pioneering their own application of grid technology. This success was soon to be introduced to the general population with the introduction of SETI@home [43] in 1999, which was replaced by the BOINC[1] software in 2005. SETI@home ran as a screensaver on normal desktop computers crunching numbers while the computer was not in use and submitting the results over the internet. [12, 22]

[1] `http://boinc.berkely.edu/`

3.1.1.1 Grid Definition

The term "grid" has its origin in the comparison of cluster computing technology to the electricity "power grid" [15, 19]. Following on from this comparison it was suggested that the grid architecture should make obtaining resources as simple as obtaining electricity from a power socket. However, in the early days of grid computing there was much debate, and many proposed definitions, of what a grid was or was not. [13, 18]

In 1998 Carl Kesselman and Ian Foster in the first edition of "The Grid: Blueprint for a New Computing Infrastructure" defined a grid as:

> "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [13, 15]

3.1.1.2 The Grid Identification Checklist

Due to the extensive debate on what exactly defined a grid, Foster in 2002 [13] drew up a checklist to identify whether or not a system is in fact a grid. Although this checklist [13], reproduced below, still leaves room for debate, it is a concise way of categorising a distributed computing system.

A grid is a system that:

- ... coordinates resources that are not subject to centralised control ... (A grid integrates and coordinates resources and users that live within different control domains for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system.)
- ... uses standard, open, general-purpose protocols and interfaces ... (A grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorisation, resource discovery, and resource access. As discussed further below, it is important that these protocols and interfaces are standard and open. Otherwise, we are dealing with an application-specific system.)

- ... delivers non-trivial qualities of service. (A grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts.)

3.1.1.3 Virtual Organisations

A virtual organisation (VO), is a set of individuals and/or institutions with some common purpose or interest and that need to share their resources to further their objectives. These resources are typically computers, data, software, expertise, sensors, and instruments [49]. Each institution has complete control over each resource it shares on the grid with this control extending to: scheduling of when the resource is available; data available on the resource; and processing power available [15, 16].

3.1.2 Grid Categorisation

Clabby Analytics categorise grids into four types [9]:

- **Compute Grids** These are the grids designed for exploiting unused computing power or CPU cycles. They have been in use for scientific, engineering, and space research for a long time.
- **Information Grids** These grids are more like peer-to peer services, primarily for the purpose of collaborative computing, e.g., file sharing. These are also sometimes called data grids, which provide standards based federated data sharing for business applications.
- **Service Grids** This type of grids combines the physical elements of grid interconnection (high speed, fabric-like network interconnect) with web services program to program architecture to deliver an environment that allows different applications, running on varied operating environments, to run and interoperate.
- **Intelligent Grids** These grids consist of basic grid network interconnect elements combined with systems/storage/network management hardware/software enhancements (and maybe even applications and database management capabilities) that enable grid devices to manage themselves or other devices on the network automatically.

3.1.3 Globus Toolkit

3.1.3.1 Description of Globus Toolkit

The Globus Toolkit is an open source grid programming framework, developed by the Globus Alliance, that implements many of the current grid standards and architectures in Java [18]. A growing number of projects and companies are using the Globus Toolkit to unlock the potential of grids for their cause. Advanced Resource Connector, introduced by Nordugrid and used by CERN [34], is an open source middleware created using the Globus Toolkit.

3.1.3.2 Globus Features

The Globus Toolkit implements the standards listed below. [14, 18]

- Open Grid Services Architecture (OGSA)
- Open Grid Services Infrastructure (OGSI) – originally intended to form the basic plumbing layer for OGSA, but later superseded by WSRF and WS-Management.
- Web Services Resource Framework (WSRF)
- Job Submission Description Language (JSDL)
- Distributed Resource Management Application API (DRMAA)
- WS-Management
- WS-BaseNotification
- SOAP
- WSDL
- Grid Security Infrastructure (GSI)

The Globus Toolkit also has implementations of the OGF-defined protocols. [14, 18]

	Resource Type	**Protocol**
1.	Resource Management	Grid Resource Allocation and Management Protocol (GRAM)
2.	Information Services	Monitoring and Discovery Service (MDS)
3.	Security Services	Grid Security Infrastructure (GSI)
4.	Data Movement and Management	Global Access to Secondary Storage (GASS) and gridFTP

Table 3.1: Globus Toolkit OGF protocols

In addition to the wide range of standards and protocols listed above, the Globus toolkit also provides a number of other libraries and software that provide security, information infrastructure, resource management, data management, communication, fault detection, and portability [14]. These provided libraries can be used independently or together to develop grid-based applications in Java. The Globus toolkit has an open source strategy, similar to that of Linux operating systems, to encourage broader use and more rapid adoption of the technology, as well as community contribution to the project [18].

3.1.4 MiG

3.1.4.1 Description of MiG

The Minimum Intrusion Grid (MiG) is a re-designed grid computing platform with the goal of providing grid infrastructure where the requirements on users and resources alike is as small as possible i.e., minimum intrusion into user and resource systems [48]. Although MiG strives for minimum intrusion, it also seeks to provide a feature rich and dependable grid solution.

MiG is driven by a stand-alone approach, rather than integration with existing systems, which allows it to be more flexible than older systems, such as Globus, that require backward compatibility and compliance with a vast range of standards [47].

3.1.4.2 MiG Features

MiG was designed as a full grid implementation with its core features being that it allows grid execution on resources without revealing the user's identity to the resource and

requires minimum intrusion on both the user and resource side. Additionally, MiG is highly scalable, has a high fault tolerance, and features good load balancing to distribute executed jobs amongst available resources [46]. MiG requires very few dependencies on either the user or resource machine, and automatically updates any required software on both. Furthermore, MiG is fully firewall compliant, with a strong adaptable scheduling engine that operates on the grid level. Finally, MiG supports banking and accounting as well as cooperative support that allows shared access to user-defined data-structures. [47]

MiG removes the single point of failure inherent in other current grid systems, such as Globus-based grids like NorduGrid, and allows for full redundancy [48]. In addition, the current grid systems all rely on job-to-resource mapping; however MiG allows for the use of real-time online-scheduling algorithms to be used to allow for maximum utilisation of all resources connected to the grid. High scalability is another factor where MiG shines over the other grid systems, and one which has proved to be a problem in one of the largest grids, NorduGrid [46]. The basic requirements to connect to a MiG system as either a client or a resource is that the connecting party has a signed x509 certificate that is trusted by the grid, and the ability to create outbound HTTP or HTTPS connections. In addition, a resource is also required to accept incoming SSH connections in order to receive jobs for execution. [47]

3.2 Grid Security

3.2.1 Introduction

Security is one of the utmost factors of importance when dealing with grid computing [8]. If the grid system is compromised, then so are all the servers and any attached resources; it is even possible that the clients that connect to the grid to submit jobs and receive results could become compromised. Thus, it is imperative to keep access to the grid and its systems under strict control, only allowing access to known and approved entities.

3.2.2 Public Key Security

Public key security involves the creation of a public key digital X.509 certificate signed with a digital signature created by a trusted certificate authority. This public key digital certificate is used to verify that the organisation or individual does indeed own the

corresponding private key and thus, is who they claim to be. Thus, the identity of an individual or organisation can be verified by providing this personal digital certificate to the server upon request. If the server accepts the certificate and recognises the certificate authority that created the digital signature as having authorised access, the client will receive an authentication challenge from the server. This authentication challenge is encrypted using the public key provided in the certificate and the client is required to decrypt it and send it back to the server as proof that the corresponding private key is in fact held by the client.

However, there are several architectural problems with the way public key security is constructed. The entire public key architecture hinges on the reliability and security involved in the signing of certificates by the trusted certificate authority [51]. If the certificate authority does not have strict guidelines that are rigorously followed when verifying identities before signing certificates, it is possible that some certificates may be signed where the recipients of the certificate are not who they say they are [42]. In addition, the certificate authority needs to have top notch security and to protect its private key used for signing certificates from disclosure or theft. If an external party gains access to this private key, the external party could sign any certificate and it would verify correctly against the certificate authority. In the recent past there have been known cases where a certificate authority's security has been breached, such as the cases of Comodo[3] and DigiNotar[2], which brings the explicit trust placed in the certificate authorities into question. It is for this very reason that many grid providers run their own certificate authorities, thereby reducing the risk of third party negligence causing a security breach on the grid system. However, this grid specific CA is not trusted implicitly by browsers and the root CA certificate will need to be imported into the browser, otherwise the client will see security warnings that the CA is not trusted. In addition, this CA root certificate will need to be placed on its own extremely secure server with physical access controls, TEMPEST shielding, and "air wall" network security [42].

3.2.3 Secure Socket Layer

Secure Socket Layer (SSL) and its newer version Transport Secure Layer (TLS) are cryptographic protocols that provide a secure communication channel between two parties over the Internet. TLS includes the ability to downgrade the connection to SSL 3.0 thereby weakening security but enabling connectivity with clients that do not support TLS [10]. SSL and TLS provide a secure communications channel by using symmetric

encryption using RSA public and private keys to encrypt data above the transport layer to ensure that no party, other than the intended one, can read the enclosed message [17]. To prevent third parties from performing man in the middle attacks and snooping, traffic asymmetric cryptography is used to perform key exchange between the two communicating parties. This asymmetric cryptography requires each party to have a valid public key digital certificate signed by a trusted CA [10]. The party requesting to initiate the secure connection, the client, connects to the server and informs the server of all its supported ciphers and hash functions. The server will then select the strongest cipher and hash function it supports and inform the client of this decision as well as sending the client its digital certificate. The client may then confirm that the server's digital certificate is valid by contacting the CA with whom it was signed. Then the client must generate a random number and encrypt it using the public key contained in the digital certificate provided by the server and send the result to the server. This encryption using the server's public key means that no third party can gain access to this random number as they would require access to the server's private key in order to decrypt it. Now, both the client and server can generate a session key from the random number, which is subsequently used for encryption and decryption of traffic sent over the wire. [27]

3.2.4 Web Vulnerbilities

Since web technology is used to allow clients to perform operations on the grid, the grid system needs to be secured against any web vulnerabilities to ensure that authorised users of the grid cannot access parts of the grid they are not authorised to. Some common web vulnerabilities are: SQL injection, cross-site scripting (XSS), and cross-site request forgery (XSRF). Grid services that use web pages to display content and allow authorised users to interact with the grid in this form need to ensure that their sites are protected against such attacks. Prevention of such attacks will greatly improve the security of the grid and reduce the risk of unauthorised activity occurring on the grid.

Chapter 4

Current State of Grid Management

Chapter 3 presented the current state of grid computing, however it did not cover how these grid systems are managed by the system administrator. Managing systems as complex and intricate as grids can be very challenging especially if the grid is managed by multiple parties across the globe. It is therefore important to investigate how these systems are currently managed in order to understand whether the current solutions are adequate and how they can be improved.

This chapter investigates the state-of-the-art and capabilities of grid management systems currently available.

4.1 Introduction

Grid management is the process of ensuring that available grid resources are used efficiently and effectively to accomplish the desired goals and objectives. Effective and efficient management of a grid involves the following: deciding who can run jobs and on which servers; how jobs should be allocated to waiting servers; and how the resources and user network should be divided. To make these difficult management decisions it is first necessary to obtain as much information as possible on the state of affairs within the grid and the servers, otherwise, decisions detrimental to the grid could be made.

Keeping a grid running efficiently by using all the resources to their full potential is key to running an effective grid. In addition, achieving this means that there should be no significant downtime. This creates the problem of either having to employ multiple system

administrators, such that at least one of them is always available to fix the problems arising from grid operation, or risking downtime; neither of which are ideal solutions. Using multiple system administrators generally means that no system administrator knows exactly what is happening on all parts of the grid at any given time, and thus when fixing a problem a system administrator may cause a regression on a previously fixed problem or introduce a new problem.

4.2 Grid Management

A grid, like most other pieces of software, generally has a number of configuration options that can be changed to alter the way the grid software operates. Most grids provide at least one method to alter these configuration options, such as a custom interface or simple configuration files that can be edited. Apart from these basic configuration options, a grid generally provides mechanisms to start and stop services or the Grid software itself.

4.2.1 MiG

The Minimum intrusion Grid [47] software provides a management interface via HTTP over SSL, which primarily allows the user to submit jobs to the grid and control resources owned by the user. Access to this management interface is restricted to users that have been authenticated by SSL using a signed certificate that is trusted by the grid. However this interface is not designed for a grid or system administrator as it provides no data on what is happening on the server, and only provides feedback on the status of jobs submitted by the logged in user.

MiG provides an administrator interface via the command-line and multiple configuration files through which server settings can be changed. MiG also provides some basic usage statistics and can be set up to provide basic graphs over HTTP using CouchDB[1] to store the data.

4.2.2 Globus

Globus, being a Grid framework and not an actual grid system, does not directly provide any management interface. However, it does provide modules that allow system administrators

[1] http://couchdb.apache.org/

to manage grid resources, as well as a module for monitoring the quality of service delivered by the grid to clients [18]. Globus also provides multiple modules for grid execution and grid data management giving the server excellent flexibility in protocol usage and job scheduling.

4.3 Server Management

Managing a server means verifying that all the required services are working on the server at all times, and performing necessary upgrades and patches to ensure that the server stays secure and functioning correctly. These operations also include performing any configuration changes that are required.

Apart from the tools provided by the grid software, there are other ways to manage grid operations. These methods generally involve management of server services such as firewalls and Domain Name Services (DNS). Since the Grid may rely on one or more of these services it is important to understand how these services are managed.

4.3.1 Console Management

Console management of services is the oldest form of computer management and one of the most widely used especially by the UNIX operating system. This type of management involves the use of a shell or command prompt and requires that the system administrator thoroughly understands the operating system as well as the specific commands for each of the services that need to be managed. This is the preferred method of management for administrators with this advanced knowledge as generally the commands offer a wider range of management options, via command-line switches, than GUI approaches. However, with the proliferation of operating systems and services it is becoming increasingly difficult to be proficient in all the required software.

4.3.2 GUI Management

The introduction of GUI management of services has significantly reduced the basic knowledge required to be a system administrator by abstracting away the underlying software running the service. This means that a system administrator no longer needs to

know exactly how the underlying software operates, but rather how the service operates as the GUI interface handles the configuration of the software. However, as stated in *Section 4.3.1*, this reduces the options available to the system administrator as it is unlikely that the GUI can provide all possible options and configurations.

4.3.3 Web-based GUI Management

The GUI approach to management covers the configuration of a single piece of software. Since the GUI for each service administers only the software running that service, administration of servers where multiple services are running can be very tedious and time consuming. With the invention of HTTP and the Internet it has become much more viable to integrate the management of services into one GUI delivered via HTTP. Delivering the GUI in this fashion also allows for remote management. However, this may introduce a security risk and care must be taken so that the administrator is authenticated beyond any doubt.

There already exist solutions that use this type of management, such as Webmin [50] and CPanel[2], which integrate multiple services into one web-based GUI accessible from remote locations and which allow the system administrator to change settings and restart services on the server on which the system resides. These systems are modularly designed so that new services can be added easily by simply writing the module for the service, however, they do not allow for administration of the same service on multiple servers from within the same GUI. These solutions also lack the ability to modify or control services while they are running, as they may only change configuration files, which will require a service restart in order for the files to be re-read and loaded.

[2]http://www.cpanel.net/

Chapter 5

Research Design

In *Chapter 4* we discussed the current state of administration and management in grid computing and identified some difficulties with the current method of grid administration and management. Because grid systems are generally very complicated and comprise a large number of distributed components each of which is controlled by a different entity, a design that accounts for all the complications of grid systems needs to be established.

This chapter presents a research design for investigating an automated grid administration system that can solve the problems with the current methodology.

5.1 Introduction

Grid computing has evolved considerably since its first inception in 1994 [18]. However, no automated maintenance management tool for grid systems currently exists, which makes easy maintenance and observation of trends on grid systems very difficult. Given that maintenance of a grid system is a critical part of keeping a grid computing service running, it is vital that the system administrators have a method for ensuring that the system is operating correctly.

Because grid systems are so complex it can take a significant amount of time to understand what is not working when a problem arises on a grid. This first step of understanding the problem is the most time consuming and is often repeated whenever a problem arises. Performing a basic analysis of the problem can be scripted, with the results showing the system administrators in which sector the problem lies. Moreover, correcting this problem

may be very simple and with an intelligent system could be solved without the system administrator's presence. Such a system would dramatically decrease the workload of a grid system administrator and become an invaluable resource in debugging and resolution of grid faults. Although the failure of the Amazon EC2 cloud system in 2011 was not due to any Grid technology fault and was, in fact, caused by insufficient redundancy on the part of Amazon, a management and repair system such as the one described in this thesis, could have significantly reduced the downtime.

However, a system such as this would require a great deal of knowledge on the innate functioning of grid systems in order to be of service to system administrators. In addition, the system should be able to detect problems on the grid with as little information as possible, so as to reduce the overhead introduced by the monitoring system on the grid, thereby freeing up more processing power for use by the grid in handling jobs.

5.2 Design Approach

To investigate the effectiveness of an automated grid repair system, the required design parameters need to be established. These parameters will assist in guiding the implementation along the correct path to maintain good research quality and desirable research outputs.

5.2.1 Fast Response Time

The most important factor when considering a repair system is how fast the repair system can identify and respond to an incident. The faster the repair system can identify an incident, the faster the repair system can attend to the problem and return the affected systems and services to a working condition. Thus, it is critical that the repair system have a fast response time in order to reduce the time that the grid system remains inoperable.

5.2.2 Security

Another important factor when considering any kind of system, especially with a system that has direct access to a grid, is the security of the system. The system should be designed with security in mind so that the system cannot easily be compromised from an

external source and each component should communicate with all the other components in a secure way. In addition, all communication between the system and the grid should be performed in a secure, encrypted fashion, thus reducing the security risk introduced by the external system to the smallest factor possible.

5.2.3 Minimising Intrusion

Although grids often include a large number of connected machines at any one time, they do not start out this way. When a new grid system is set up for the first time and made available to the computing community to use and add resources, there are initially no resources connected. Each machine that is to be connected as a resource needs to have all the software required by the grid installed in order to communicate successfully and be able to accept, execute, and return jobs. Thus, it is important that the grid requires as little as possible in terms of software to reduce the initial investment required to connect to the grid. In the same way, the repair system should require as little software as possible so that it can easily connect to grid servers and resources alike to repair any fault that occurs. The most effective method of doing this is to ensure that the repair system's software requirements for connecting to either the grid server or a resource is a subset of the software requirements of the grid itself. This means that by installing the required software to connect a resource to the grid, the software required to repair and monitor this resource is also installed.

5.2.4 Maximising Gained Information

To align with the objectives of the above paragraph, the repair system should also maximise its usage of information obtained from a monitored machine. By maximised usage of information we mean that any information gained from the monitored machine should be used to its maximum potential for determining the state, status, and serviceability of the monitored machine. This maximisation of information means that fewer commands or scripts, than if all the information was not used, need to be executed on the monitored machine to determine its state, thereby freeing up processing power for execution of jobs instead of monitoring tasks.

5.2.5 Modular

Modularity is one of the core principles of object-oriented programming and applies exceptionally well to models that are required to be easily extensible. Modularity is also significantly applicable to application models where more than one type of backend is supported. Thus, by limiting the interface to a single module, this backend can be quickly and easily replaced by another merely by changing the module used. This approach also allows for quick adoption of new interfaces as they can be created as a new module and installed seamlessly into the system. Thus, it is important that the repair system is created using a modular approach as it could be required to support more than one grid system, and this approach would allow it to support any new grid systems created in the future.

5.2.6 Cross-platform

When working with grid systems it is common to find more than one operating system in use on either the grid server or the resources. Thus, it is important for any repair system to be able to work on or with multiple different operating systems. This means that the repair system needs to be designed with cross-platform compatibility in mind and should be constructed using software that can be easily ported to a different operating system.

5.2.7 Open-source

When producing a prototype system to investigate the feasibility of an advanced system such as that of a grid repair system, it is important to know how each individual part of the system works so that the research (and ultimately the results) can be repeated. In addition to this, it is not possible to know all the functions and inner workings of a proprietary product or protocol that may perform unwanted or necessary functions thereby skewing the research results. It is, therefore, in the best interests of the research to limit any products or protocols used to open source products and protocols.

5.3 Proposed Design

To carry out the research while taking into account the design parameters and restrictions discussed above, the general overview of the prototype design needs to be outlined to guide the implementation of the prototype.

There are three basic components of the repair system, as depicted in *Figure 5.1.* The first is the logging part, which connects to the monitored machine, runs any required commands, and finally gathers and saves the output. The second component uses the saved information to detect any problems on the network or the monitored machine on which the data has been saved and alerts the third part of the system to any detected irregularities or unexpected behaviour. The third component of the system is triggered by the second and uses the data from the second part to diagnose and repair any faults that may have occurred on the monitored systems, networks, and attached systems.

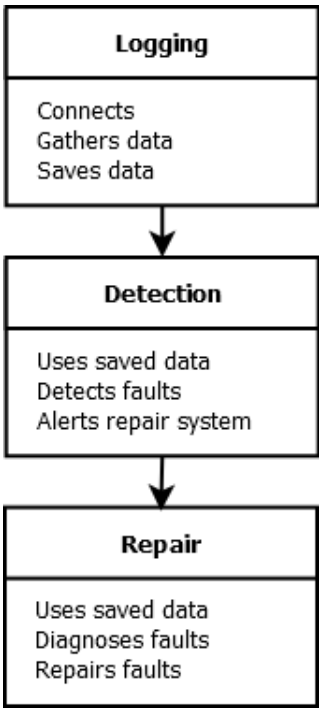

Figure 5.1: Proposed system design

Now that we have outlined a basic design for the system, we can focus on the parameters for how the research will be conducted. First, a Grid server needs to be installed in a virtual machine and configured correctly. Then, the prototype system needs to be created according to the specifications described above and set to monitor the virtual machine on which the Grid server resides. Once sufficient data has been captured, the data must be analysed to observe how well the prototype system has performed and what effect it has had on Grid uptime, performance, and productivity.

It is expected that the results will show improved uptime of the Grid owing to quicker recovery from any errors or faults. This then implies improved Grid productivity and reliability as a result of this improved uptime. However, there may exist cases where the repair system does not have sufficient knowledge or access to repair the fault that

has occurred. In these cases the repair system may need to be given access to systems outside of the Grid infrastructure in order to repair them, or be trained further with more diagnostic tests and repair procedures to repair the Grid system completely.

5.4 Summary

Grid computing has evolved significantly over the last decade with the introduction of cloud computing. Nevertheless, no management tool or automated tool for monitoring and repairing a grid currently exists. Thus, by creating a prototype of such a system and testing its effectiveness, we can determine whether such a system could improve the grid computing sector. Some key factors need to be considered when creating the prototype, including response time, security, modularity, and system intrusion. It is expected that such an automated monitoring and repair tool would significantly improve grid system uptime and productivity.

Chapter 6

Implementation

This chapter presents an overview of the implemented system used to carry out the research on automated grid administration systems, which could solve the problems with the current methodology as described in *Chapter 4*. This implementation conforms, as closely as possible, with the research design ideals described in *Chapter 5*.

6.1 Introduction

A prototype system was implemented to investigate the feasibility of the research design ideals described in *Chapter 5*. The reasons for selecting MiG, as described in *Section 3.1.4*, as the grid of choice for creating the prototype are twofold: its limited intrusion on remote systems and definite security enhancements compared with the available current generation free-to-use grids. In addition, MiG was fairly easy to install on the recommended operating system, Debian 5, and base configuration files for all the needed services were included to get the grid server working quickly. This made initial testing and concept prototyping a success and allowed for quick progression of the project into the development phase. MiG support was also quickly forthcoming from the developers to overcome any installation problems encountered at the time, thus making the install far easier than installing any of the other available grid servers. With development and documentation improving continually it is clear that MiG is becoming one of the forerunners in grid computing servers. MiG also includes an XMLRPC interface for sending requests directly to the server, although this currently only supports requests that could have been made via the web-interface. Nevertheless, due to the fact that it is open source and implemented in

Python, extra commands can easily be added to support advanced control of the MiG system.

Since MiG was selected as the grid of choice for testing the implemented prototype, Python was selected as the language for implementing the prototype. This was an obvious choice as MiG was developed in Python and thus if the need arose to directly integrate the prototype with MiG it would be far simpler to do than if the prototype were developed in another language. Although PHP is widely used as the web-development language of choice and provides many advantages on the web-programming front, the web-development contained in this thesis is purely for graphical feedback purposes and does not form part of the actual repair system. Thus, the great ability of PHP to perform in web-development does not lend itself to this thesis, and Python's ability in a wide range of programming spheres is far more beneficial to a system of this type. Another benefit of using Python is that it contains all the required software modules for the creation of the prototype, including any dependencies as installable add-on libraries. The fact that these dependencies were included in Python meant that there was no reliance on including external C++ programs to perform any required dependant tasks, for example creating an SSH connection to another machine.

MySQL was selected as the database of choice as it provides all the functionality required for the prototype and the Python libraries used for the prototype contain built-in support for MySQL [4]. MySQL has sufficient performance to enable effective logging of data from a multitude of monitored servers using bulk inserts, which improve insert performance significantly [35]. Insert performance is the most important owing to the high number of inserts being executed when monitoring a large number of servers. However, since the performance of select in MySQL is also good, the generation of graphs and statistics for use by system administrators is simple and fairly quick [35]. In addition, prior knowledge of MySQL operation and installation procedures lowered the learning curve required to develop the prototype system.

To effect repairs on the system, an intelligent learning ability is needed to form a hypothesis on the cause of the fault and then to attempt to repair this fault by the methods known to the system. Since a hypothesis on the cause is required, the system needs to perform a diagnostic on the grid in an attempt to determine this. After performing such a diagnostic, a hypothesis can be presented and then an attempt at repairing the system can be made. Neural and Bayesian networks, as described in *Sections 2.4.1, 2.4.2* provide excellent learning methods for learning which commands are effective in restoring the grid to a working condition once a fault has occurred. However, they are not suitable for diagnosing

a grid and providing a hypothesis as to why the grid is no longer functioning correctly. Expert systems, as described in *Section 2.4.3*, provide a very good logic base to assist in the diagnosing of the grid system and form a hypothesis as to the root cause of the fault currently being experienced. Due to their advanced logic ability, expert systems provide a unique advantage in executing repair commands on the grid as the logic diagnosis process would have ruled out some of the possible root causes thereby eliminating the need to attempt certain repairs. This means that the expert system is left with a hypothesis of the root cause and a small number of commands that could repair such a fault. Expert systems can also be trained in a similar fashion to neural and Bayesian networks by rewriting the expert system rule base and loading it fresh at each execution. Rewriting the rule base in this fashion means that the commands, or command series, that work well to repair a specific fault can be shifted up the execution order when such a fault hypothesis is proposed. Thus, expert systems have a significant advantage over neural networks and for these reasons are used in our proposed repair system. It is possible for neural and Bayesian networks to be used in conjunction with the expert system to improve the learning ability of the repair system. However, this is not addressed in this thesis, but may be considered as a possible extension of this research.

The base system was designed as a modular logging system implemented in Python that connects to the MiG server via SSH or XMLRPC to gather the required information for logging. Each module in the logging system is completely independent and is given the SSH and XMLRPC interfaces to run the commands required to retrieve the needed information. This information is retrieved by the relevant communication channel and passed on to the relevant installed module that requested it. This module then performs any required parsing or processing on the data before storing it in the MySQL database. Each module is able to create its own table in the MySQL database and as such, can specify the structure of that table, which is created the first time the module is compiled and started with the logging system. Thus, there is no need to have a generalised table format in which modules can save data, as each module can specify the way it wants to save its own data. Once all modules have logged their data for the required polling interval, the data is passed through each module's detection algorithm in turn. These detection algorithms take into account the respective module's assumptions and attempt to detect any irregularities within the data. If any irregularity or problem is detected, the data is passed to the Pyke expert system for analysis. The Pyke expert system imports each modules rule base and attempts to diagnose the possible fault by checking the grid system for errors and problems. If any errors or problems are encountered during diagnosis, the Pyke expert system will come to a conclusion about what the root cause of said errors

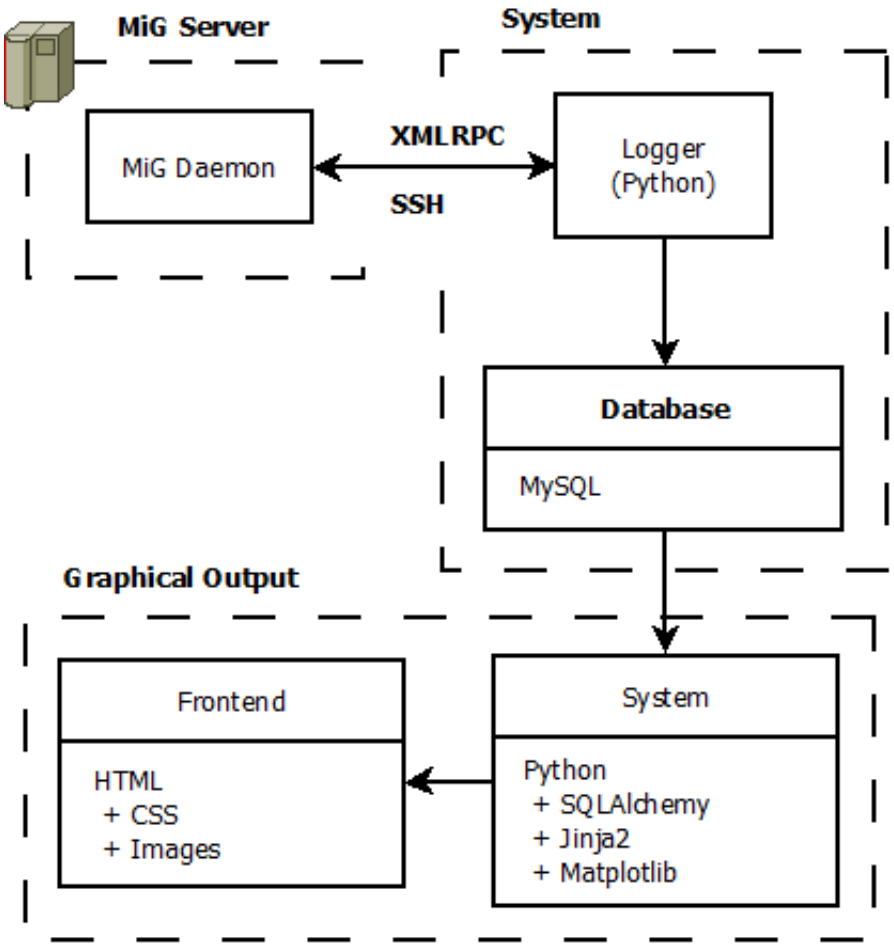

Figure 6.1: General System Overview

and problems is, and then attempt to correct it by running the appropriate commands.

Once all the logged data is safely stored in the MySQL database, the information becomes available to the front-end system for viewing by system administrators. To output graphs and statistics for use by the system administrators, Python is used to generate HTML pages that can be viewed by the system administrators to ascertain what is happening on the network and monitored systems. As shown in *Figure 6.1*, multiple Python libraries are used to perform the generation of HTML pages and graphs, thereby simplifying development, as well as producing a plausibly deployable prototype. No specific security code has been put in place in the prototype, however base security comes from RSA key authentication, which is enabled on the Apache server. In addition the Python modules used have further security measures built in, such as dual data channels connected to the SQL database where one channel sends the SQL statement and the other sends the required data thereby eliminating the possibility of SQL injection.

6.2 Implementation Challenges

In the implementation of the prototype, as described above, certain challenges had to be overcome with respect to the implementation of the design ideals as described in *Chapter 5*. These implementation challenges arose mainly as a result of the implementation of the design ideals that either did not go as planned or had to be adjusted to work correctly.

6.2.1 Modular Plug And Play Design

One of the design ideals is that the prototype system should be created in a modular manner to facilitate the addition of new modules to extend the monitoring and repair feature set. Although such a modular design is in theory a good idea; implementation of such a design element is complicated when dealing with a system that needs an exceptionally high uptime to monitor a grid system continuously. To mitigate downtime when a new module is installed into the prototype, the prototype must be manually stopped and restarted thereby forcing a system administrator to be present in the case of any failures. In addition the prototype does not directly include new modules that are installed, but instead searches for all available modules on start-up and dynamically compiles them before including them. The dynamic compilation of modules means that if a module fails to compile it will not crash the entire system, and only that module will not be loaded. Since the prototype does not require direct inclusion of new modules, no code needs to be changed in the prototype code base when installing a new module into the system.

6.2.2 Securing The Repair System

Another of the design ideals is that of security, however once again this is something not always easily implementable as there are consistently new ways of bypassing and hacking the security measures in place. However, in this case a best effort has been made to secure the repair system from any unwanted intrusions into either the repair system itself or the grid. Communication between the repair system and the grid process is conducted using XMLRPC with RSA public-key host authentication. Communication between the repair system and the machine itself is conducted using SSH with RSA public-key host authentication. In addition to this, the web-based front-end for the repair system has no communication link with the repair system itself and only connects to the database to read data through a database user with no write access. Thus, most of the risk for a

breach occurring within the repair system is mitigated, although it is, in fact, shifted onto the RSA public-key authentication. This means that if the host machine is compromised in another way or the RSA private keys are leaked, the security will be compromised. This single point of failure is not ideal and could be mitigated by employing multi-factor authentication with the grid process and host machine.

6.3 Implemented Metrics

This section provides a short description and overview of how each of the metrics was implemented to allow the reader to reproduce, if desired, the experiments and results as listed in the subsequent results chapters.

6.3.1 RAM Metric

The RAM (random access memory) metric is based on the fact that when something happens on the server there is likely to be a significant change in the memory usage of the system. An example of this is when a process crashes; the operating system terminates the process and then removes it from memory freeing all the memory the process was holding. A process may also be using space in the swap file; however, the RAM metric implementation takes this into account by checking for a corresponding change in the swap file when there is a drop in memory usage. This is accomplished by checking for a similar size change on both the swap file and memory usage within a pre-defined margin of error. This margin of error is due to the fact that not all of the memory allocated to a process may actually be in use by the process at any one time. Thus, an event is not triggered if a process has just been moved from RAM to the swap space, or vice versa.

To reduce the search parameters for the RAM metric, some basic assumptions were made regarding the use of the metric in detecting events. Firstly, it is assumed that the machine monitored by the RAM metric is not a development machine; i.e. there will not be continuous service or server restarts for modified code to be brought into use and tested by developers. Thus, since there will not be intentional server or service restarts it is expected that the RAM usage of the monitored machine should stay fairly constant. Thus, it is expected that the RAM usage of the monitored machine will only have minor variations over time as services require more or less memory to process requests. These assumptions were taken into account when developing the event detection mechanisms for the RAM

metric and thus, if proved false for any monitored machine, would significantly reduce the effectiveness of the RAM metric on such a remote machine.

The event detection on the RAM metric operates by watching the overall memory usage of the system for a significant change and is implemented as described below. First, a rolling mean is calculated using a given window size. Then, a simple search algorithm is used to find the points of change within the rolling window; if the difference between any two points is greater than five percent of the calculated rolling mean between the lowest and highest values, a change (event) is signified. Now to ensure that the located points of change are statistically significant, a rolling window is generated for the window up to, but not including, the change point. If it is found that the change point falls outside of the expected data zone, which is the mean of the window plus or minus five percent of this mean, then the change point is listed as a fault. This method will detect a fault accompanied by either a large increase or decrease in memory usage.

The idea behind using the window up to, but not including, the change point is to increase the reaction time of the system. Since the RAM metric data is continuous, it is known that the next data point should be fairly similar to the previous and successive data points. This is easy to test for old data points, where data points exists on either side of the tested data point. However, as the data points come into the system in real-time, only older data points exist against which to test. Therefore, by using this method of testing, the system's reaction time is reduced to the time to produce one new data point. This means that when an event occurs the system will be able to react in a time less than or equal to the polling interval.

If an event is detected, the system first runs a series of tests to determine if the detected event is in fact caused by a fault or crash in the grid software. This is done since a server is commonly responsible for multiple services at the same time, some of which may include grid services and software. Since the RAM metric is so generic it is quite possible for an event to be detected when a failure of another service running on the same server occurs. Once it has been determined that it is indeed a grid service failure that caused the event, the system collects a series of facts reflecting the status of the required services and state of the server. The collected facts are collated and passed on to an expert system, which using the given rules can devise the correct series of responses and commands to be executed on the system to rectify the grid service failure.

6.3.2 Grid Jobs Metric

The grid jobs metric is based on submission, execution, failure, and completion of grid jobs. By observing these grid job events it is possible to optimise grid scheduling and protect the grid from being used by unauthorised parties. For the purposes of implementation and testing, the metric has been implemented to understand and interface with the MiG server. This metric is, however, by no means bound to this grid middleware and can easily be adapted for any alternate grid middleware that offers the same set of functionality.

The event detection on the grid jobs metric operates in a similar fashion to the RAM metric in that it monitors for large changes in usage, such as sudden spikes in job submissions. However, the grid jobs metric is far more complicated than the RAM metric as there is no constant data. This means that the job events have to be grouped into time intervals to make sense of them. Moreover, the time interval used for grouping needs to be carefully selected as it can drastically alter the results of the metric.

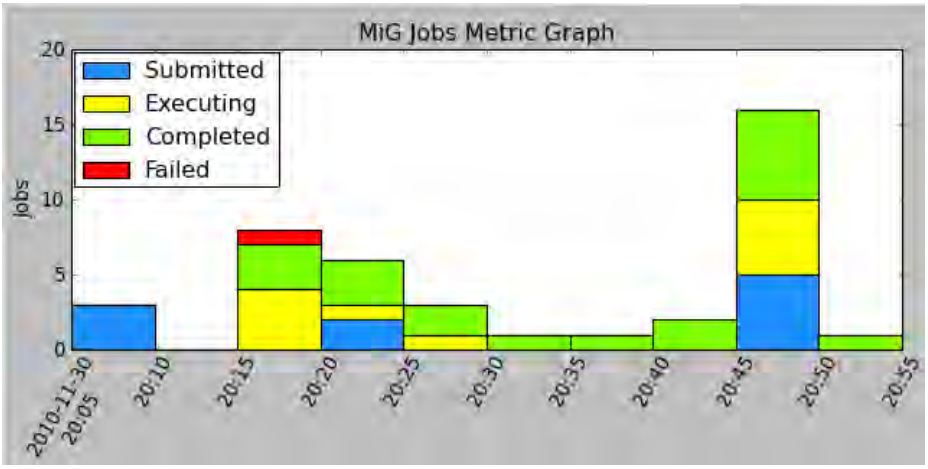

Figure 6.2: Grid jobs metric - example graph

To compensate for the absolute dependency on grouping interval when evaluating the grid jobs metric data, multiple evaluation passes are required. Each of these passes uses a different evaluation interval so that the data from varying intervals can be overlaid in an attempt to locate inconsistencies or problems with the grid jobs processing. This will stop the metric from triggering an event when the data are in fact consistent, but the selected evaluation interval has grouped them poorly. However, this results in a significant trade-off in terms of the computer power required to identify problems using the grid jobs metric data with this overlaying method.

An alternative to the methodology expressed above of viewing the grid jobs metric data as discrete data, is to view the grid jobs events as a continuous data set. This can be done

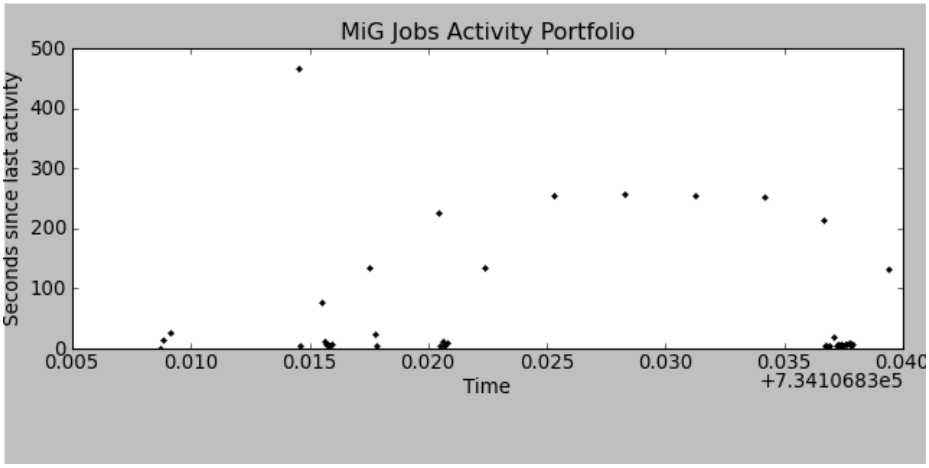

Figure 6.3: Grid jobs metric - activity portfolio

by considering each job event as it occurs in real-time instead of grouping them using a specific time interval. This method is displayed in *Figure 6.3* where each job event is shown at the exact time it occurs, with the height showing the number of seconds since the last job event occurred. This graph gives far more information than that illustrated in *Figure 6.2* for this data set. It can be seen that for this data set the job events are reasonably spaced with some groupings where job events occurred in quick succession. This information is not made obvious by *Figure 6.2* as all that can be ascertained from it is that the number of events displayed occurred somewhere within the interval of the respective bar.

Although *Figure 6.2* provides some very valuable information, it is not exceptionally useful as either a visual tool for system administrators or a dataset that can be mined easily by an automated system. So to make this plausible for these purposes, the data needs to be transformed into a more usable form. This transformation can be done by ranking each point according to how many job events are in the vicinity thereof. With this kind of ranking and only using event data points older than the current event data point, it can quickly be established when a high number of job events are occurring within a short period.

This ranking approach is shown in *Figure 6.4*, where each job has been ranked by the number of jobs in the interval preceding that in which the point is located. From the graph it is clear that a large number of job events occurred within a very small interval around 20:48. By referring to *Figure 6.2* it can be seen, that in this case it is because of the high number of job submissions, executions, and completions that occur within a five minute interval; however, this may be an early indication of a problem or high load on the

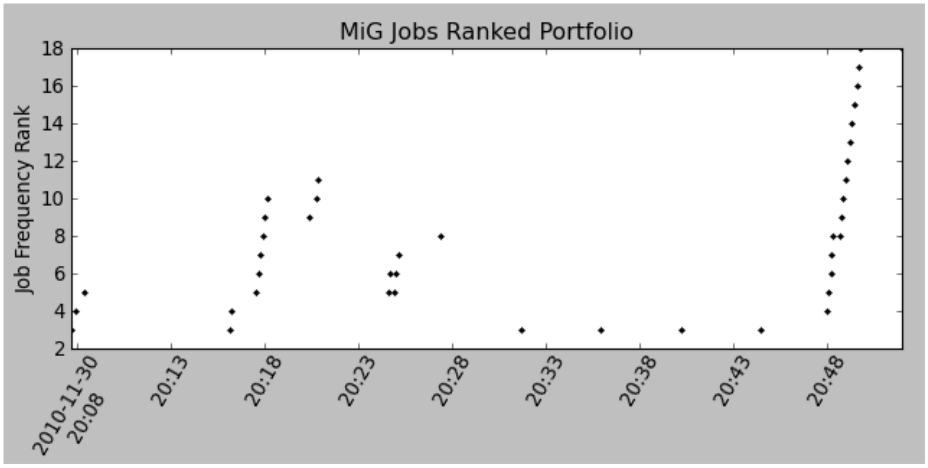

Figure 6.4: Grid jobs metric - frequency ranked

grid. *Figure 6.4* does not, however, provide an easily readable plot and this could be much improved by converting it to a line plot by connecting the plotted points. This line plot can be further improved by using the event data to construct a new data set of evenly, closely spaced points where each point is ranked using the same approach as before, that is, by calculating the number of event data points in the immediate vicinity preceding where the data point is located. Plotting this new data set produces a much smoother line, as can be seen in *Figure 6.5*, thus allowing better visualisation of the activity on the grid than the dot plot shown in *Figure 6.4*.

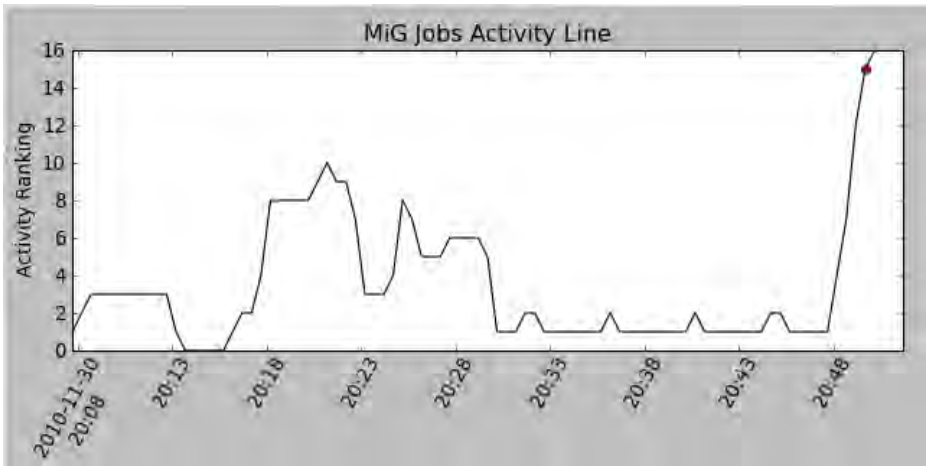

Figure 6.5: Grid jobs metric - ranking line

Figure 6.5 shows the smoothed, ranked line of the same data used to generate the bar graph in *Figure 6.2*. By inspecting the difference, it can be seen that the activity line shows a much better representation of when job events occur on the grid as it does not use a fixed interval to show when events occurred and instead shows them at the actual

time they occurred.

The event detection on the grid jobs metric operates by observing the ranked data set and attempting to locate problems and faults using a statistical approach. This statistical approach allows some leeway in changing conditions and does not trigger events when the data remains within the bounds of recent data trends, both in terms of variation and an increasing or decreasing trend. The grid jobs metric uses a rolling window approach for detection, which means that only a small portion of the most recent data set is considered when deciding if an event should be triggered. The use of rolling windows allows dynamic detection when the conditions on the grid are changing quickly and also allows better detection in the face of changing trends. Detection using statistics can give quite a broad range for detection when the data has a high variation and this rolling window approach helps the detection adapt quickly to the changing trend. Since the data comprises the total grid job events, it is expected that the data will in general conform to a normal distribution. This is due to the likelihood that job events submitted will execute in the next cycle and then be returned from the server in a completed state a short time after that since there are a finite number of resources to process submitted jobs.

Using the mean and standard deviation to do the detection over the selected window allows detection of values that exceed the expected value range; however, this approach does not readily detect periods of inactivity. Since the data is known to be skewed right, as there can be no left tail given that there cannot be fewer than zero job events in a time period, a few improvements can be made to the basic statistical detection to detect when the grid is experiencing an unusual period of inactivity. The basis of this inactivity detection switches from using statistics to using mathematics to compute where an unexpected or unusual period of inactivity is being experienced. Mathematics provides a unique ability to analyse graph functions accurately, and within this large field the specific concept needed for detecting inactivity is that of the inflection point. An inflection point is a position on the graph where the first derivative is a local maximum or minimum and the second derivative is zero [44]. This can be more easily defined in visual terms by stating that an inflection point is the position on the graph where the curve turns from concave up to concave down or vice versa. *Figure 6.6* depicts the graph of $\sin x$ showing the inflection point on the graph. It can clearly be seen that the curve on the left of the inflection point is concave up and the curve on the right is concave down.

This inflection point is of significant importance when monitoring data for inactivity, as it allows the system to determine the point at which the consistency of job events is approaching zero and thus undergoing a period of inactivity. This detection is done by

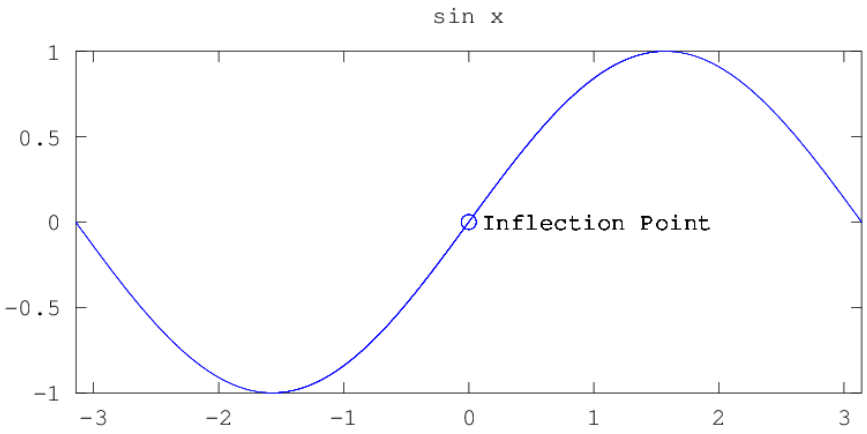

Figure 6.6: Sin(x) graph showing the inflection point

combining the inflection point theory with line smoothing techniques to obtain a good curve from the data. Following this, the distance from the last known occurrence of the expected data level to the inflection point is measured and compared to the known average distance from the data to the inflection point to determine if it is significantly greater. If the current distance to the inflection point is greater than twice the standard deviation plus the known median of the distance expected from the data to the inflection point, the grid is considered to be undergoing an unexpected period of inactivity.

The grid jobs metric uses a default window size of 61 data points and detects outlier points in a window that falls outside of the mean of the window plus two and a half standard deviations. Two and a half standard deviations are used to limit the detections of outliers in the dataset to data points that fall outside 1% of the confidence interval. To decrease the number of detections in the case of rapidly changing detections, the detection window is reset to the point immediately after an event has been triggered. Resetting the detection window in this way means that the new window includes points with only the new trend in effect. This means that the system can quickly adapt to the new trend and will not trigger events excessively during the period where the trend is changing.

6.4 Expert System

This section provides an overview and discussion of how each part of the expert system operates. It goes on to describe how the expert system attempts to find the root cause of the problem and solve this, thereby addressing other apparent problems within the grid system.

Pyke, as described in *Section 2.4.3*, was selected for use in the implementation of the grid repair system due to its vast versatility, direct integration with Python, and support for assembling functions into complete call graphs. Pyke's inclusion of backtracking is also an extremely valuable feature as the expert system implementation for the repair system uses a backward-chaining rule base. The backward-chaining rule base was selected as repairing the grid system is the obvious end-goal of the expert system. The Pyke backward-chaining rule sets also include the ability to execute arbitrary Python code, or functions, alongside the rules and it is this feature where Pyke shines in its diagnostic ability. By allowing Python function calls, the rule base can run diagnostic functions on an as needed basis instead of requiring a large amount of debugging and data gathering code to run before passing the result set to the expert system for analysis. This means that the rule base will execute only as many debugging routines as is absolutely necessary while diagnosing any grid faults that have occurred.

Listing 6.1: Simple Pyke backward-chaining rule example

```
1  # plan_example.krb
2  # Simple example plan creation
3  transfer_rule
4    use transfer($from_acct, $to_acct) taking (amount)
5    when
6      withdraw($from_acct)
7        $$(amount)
8      deposit($to_acct)
9        $$(amount)
10
11 withdraw_rule
12   use withdraw(($who, $acct_type)) taking (amount)
13   with
14     print "withdraw", amount, "from", $who, $acct_type
15
16 deposit_rule
17   use deposit(($who, $acct_type)) taking (amount)
18   with
19     print "deposit", amount, "to", $who, $acct_type
```

Listing 6.2: Running the simple Pyke example

```
1 # simpleexample_driver.py
2 # run the simple example from within Python
3 from pyke import knowledge_engine
4 # create the knowledge engine
5 engine = knowledge_engine.engine(__file__)
6 engine.activate('plan_example')
7 # create the execution plan
8 no_vars, plan = engine.prove_1_goal('plan_example.transfer((
     bruce, checking), (bruce, savings))')
9 # execute the created plan
10 plan(1000)
```

Listing 6.3: Simple Pyke example output

```
>>> plan(1000)
withdraw 1000 from Bruce checking
deposit 1000 to Bruce savings
```

Listing 6.1 shows a simple example of Pyke code, saved as 'plan_example.krb', that allows transfers between local accounts at a bank and has been adapted from the examples shown on the Pyke website [37]. *Listings 6.2* and *6.3* show how to use the example shown in *Listing 6.1* and what the output of this example is when run with *amount* set to 1000. From *Listing 6.2* it can be seen that creation of the expert system knowledge base is fairly straight forward, and the backward-chaining rule base can be loaded from the file 'plan_example.krb' as shown on line 5. Following this the expert system inference engine is invoked to find a solution that returns a valid call graph for the 'transfer' rule using the loaded backward-chaining rule base. If a solution is found, then an execution plan is created and all the parameters passed to the inference engine are hard-coded into the execution plan.

The creation of the execution plan requires that a solution be found for the required rule using the backward-chaining rule base provided to the inference engine. This is accomplished by locating all the rules that have the required rule in their 'use' clause and then recursively attempting to prove each rule's sub-goals. From *Listing 6.1* we can see that the *transfer* rule requires the sub-goals of *withdraw* and *deposit* be met before it is considered valid. Neither of the rules that have *withdraw* and *deposit* in their uses

clauses require any further rules and thus will always return as valid and each adds a print statement to the returned call graph. Thus, the execution plan that is created in line 5 of *Listing 6.2* and saved into the Python variable *plan* is in effect a Python method that will transfer the *amount* passed as a parameter to it from the account 'Bruce checking' to the account 'Bruce savings'. Thus, when the plan is called with the parameter 1000, the resulting output is as shown in *Listing 6.3*.

Listing 6.4: Pyke code for repair of repair system front-end

```
# repair_frontend.krb
# Pyke bc rule set for repairing front-end
restartfrontend
  use restart_frontend($serverdata)
  when
    # attempt to connect to the server on port 80
    check portconnect($serverdata.ip, 80)
    # check the system for the repair front-end process
    check noprocess($serverdata, 'frontend.wsgi')
    # if the above passes, attempt to start the frontend
    start_frontend($serverdata)

startfrontend
  use start_frontend($serverdata)
  with
    # attempt to repair the frontend
    repair_frontend_process($serverdata)

bc_extras
  import sys, socket
  import StringIO
  import subprocess, signal, os
  from datetime import datetime

  def portconnect(ip, port):
    # setup a TCP IPv4 socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
      # attempt to connect to 'port' on remote machine
```

```
      s.connect((ip, int(port)))
      # if connect did not fail, shutdown the connection
      s.shutdown(2)
      return True
    except:
      return False

def noprocess(server, procname):
  # get the output of 'ps -A'
  p = server.ssh.execute('ps -A')

  # run through the output of 'ps -A' looking for a python
     process
  for line in out.splitlines():
    if 'python' in line:
      # if we find a python process grab the process ID
      pid = int(line.split(None, 1)[0])
      if pid != os.getpid():
        # check the '/proc/' and read the cmdline arguments
            for the process
        f = server.ssh.execute('cat /proc/{}/cmdline'.
           format(pid))
        # check for the required script name in the cmdline
            arguments
        if procname in f[0]:
          # return false if the correct process is found
          print "grid process found"
          return False
  print "no process found"
  return True

def repair_frontend_process(server):
  # start the repair frontend
  print datetime.now(), "starting grid process..."
  pid = server.ssh.execute('nohup python frontend.wsgi')
  return True
```

Listing 6.4 shows the code that restarts the repair system's front-end solution that provides system administrators with graphs and statistics. In the same fashion as the simple Pyke example shown above, this backward-chaining rule base is called with the end goal of repairing the repair system's front-end process. Thus, execution of the backward-chaining rule set will start at the 'restartfrontend' rule that has *restart_frontend* in its use clause. If port 80 on the target server is not connectable and the front-end process is not found in the output of 'ps -A' then the rule set successfully completes and returns a plan that can be executed to start the Python front-end process on the target server. Unlike the simple Pyke example shown above, this backward-chaining rule set uses real Python methods listed in the 'bc_extras' section to dynamically interact with the grid systems and servers to diagnose the exact nature of the problem as the inference engine executes the rule set. This allows the rule set to return a plan that uses a method, or combination of methods, that will successfully restore the grid systems to a working condition.

6.5 Metric Security

This section provides an overview and discussion of how each of the metrics improves the security of the grid.

In addition to detecting events and problems on the grid system, the metrics can improve the security of the computer running the grid services. This is accomplished by observing the metrics on the server in a similar fashion to what is required when detecting problems in order to identify anomalous or unexpected behaviour.

6.5.1 RAM Metric

The RAM metric monitors the random access memory of the grid server, looking for anomalous activity such as large memory spikes and dips. When such anomalous activity is detected, the RAM metric attempts to determine if this activity was caused by the grid services. It does this by passing the relevant information to an expert system. If the expert system finds no fault in the operation of the grid service then it is highly likely that another process has started or stopped, which might affect the security of the system.

When identifying events where the expert system can find no fault with the operation of the grid services, the system should alert the administrators to a potential problem with

the grid. There could be a set of conditions causing the grid to function incorrectly, but which the logic of the expert system is not powerful enough to identify. Such a warning allows the administrators to inspect the system and determine if any unauthorised process is running or any important non-grid related process has crashed. It is important to know when such events occur as grid servers are extremely complex and could easily have unknown security vulnerabilities that allow an intruder access to the server. This intruder could then start running processes in the background and effectively take control of the server.

6.5.2 Grid Jobs Metric

The grid jobs metric monitors job submissions, executions, failures and completions. It also monitors any unexpected changes in grid usage. When anomalous activity is detected, the grid jobs metric attempts to determine if this activity was caused by a fault with the grid services.

If anomalous activity spikes are detected, but cannot be attributed to either normal grid operation or a fault on the grid system, then it is possible that the grid middleware has been compromised and a remote intruder is injecting jobs into the grid without authorisation. On the other hand, a sudden dip in grid jobs activity may indicate an unknown problem with the grid middleware that may have been caused by a remote intruder attempting to break into the grid system, causing it to malfunction and crash.

6.6 Summary

This chapter has described an implementation of the grid repair system in sufficient detail that the reader will be able to reproduce the results presented in *Chapters 7* and *8*. In addition, it has shown that care has been taken to take into account all the factors affecting grid computing systems in the implementation, while adhering as best as possible to the design ideals as stated in *Chapter 5*.

Chapter 7

RAM Metric Results

This chapter presents and explains the experiments conducted on the RAM metric, as described in *Section 6.3.1*. It then goes on to present the results of these experiments and a discussion thereof. *Chapter 8* presents and describes the experiments conducted on the grid jobs metric, as described in *Section 6.3.2*. *Chapters 7* and *8* both comprise of multiple scenarios where the metric in question was tested to ascertain if it would detect the event occurring the in scenario. After the test is explained and conducted, the results are presented with a discussion as to the effectiveness of the metric in detecting the tested event. Following these scenario chapters is an evaluation chapter whereby the results given in *Chapters 7* and *8* are critically analysed as to their performance, accuracy and effectiveness.

The data used to obtain the results given in *Chapter 7* were created from a privately run MiG server, which was associated with a single resource and loaded with jobs as necessary. The data used to obtain the results shown in *Chapter 8* were obtained from the MiG project server that was run for an HPC (High Performance Computing) course at the Datalogisk Institut of Kobenhavns University, Denmark in 2011.

7.1 Introduction

As described in *Section 6.3.1*, the RAM metric is based on the assumption that when something happens on the server there is likely to be a significant change in the memory usage of the system.

For testing purposes, one minute data capture intervals are used in all the scenarios given below, except where explicitly stated otherwise, so as to retain consistency of the results. The system uses a one minute interval as this allows fairly fast detection of a problem. In general, detection should occur in less than two data capture intervals and should not be exceptionally intrusive on the observed systems, as would be the case with a data capture interval of one second. In other words, there should not be any effect of the data capture overhead on the results.

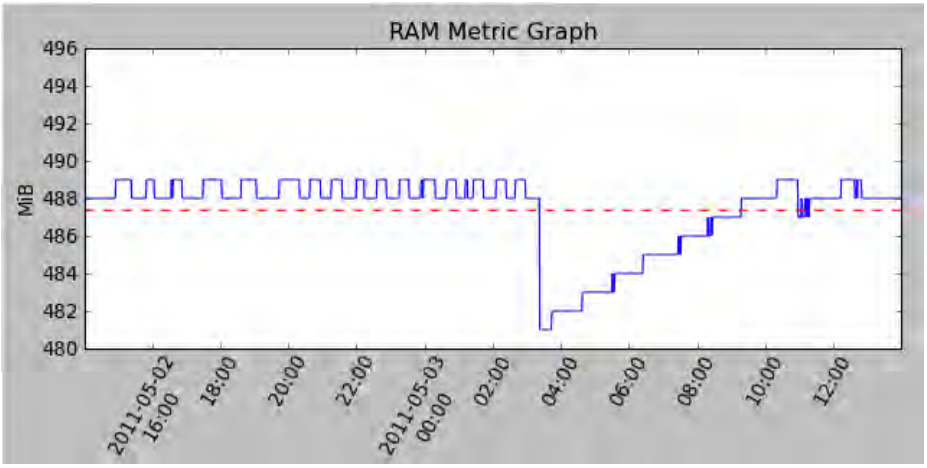

Figure 7.1: RAM metric - general usage data

Some general capture data can be seen in *Figure 7.1*, where the Grid server has been running for a while and the memory usage has stabilised. We would expect to see this type of graph if all services are working correctly on the Grid server and no problems have occurred within the last 24 hours. The graph shows that a process cleaned up some of its memory at about 3 AM and then continued to consume more memory. This memory dip is most likely a result of Python or Java running their garbage collection routines and does not significantly affect the memory with only a change of 7 megabytes.

To be able to interpret the graphs for the RAM metric it is important to know what each of the lines depicted in the graphs means. The solid black line at the top of the graph shows the total memory usage of the machine, which should remain constant unless the machine is turned off and more memory added to it. The solid blue line represents the captured data from the monitored machine; i.e., this is the amount of memory in use at the time of capture. The dotted red line is the mean of the captured data for the shown interval; that is, it is expected that 95% of the data should fall within two standard deviations of this line. Finally, red dots can be seen on some of the later graphs in this chapter showing where a system irregularity has been identified and an event triggered.

7.2 Sudden Memory Change

7.2.1 Scenario

One of the most basic scenarios that should be caught by the RAM metric is a sudden large change in memory usage. As described in *Section 6.3.1*, this is the basic use case for the RAM metric and detects large memory dips, spikes, and troughs.

7.2.2 Expected Results

Since this type of detection is not necessarily due to a fault or service failure on the server, it is expected that there will be cases where an event will occur without a fault existing (false positive). There is also the possibility that an event is not identified, even though a fault or service failure did occur on the server (false negative). The risk of these false-positives and false-negatives can, however, be mitigated by decreasing the confidence value with which the RAM metric detects events.

7.2.3 Achieved Results

As described in the expected results, the meaning of the achieved results for this scenario are quite vague and events that occurred may not necessarily have been related to Grid operation. However the event detection was very successful in detecting when the scenario occurred within the dataset, firing off an event response within one to two minutes of the event having occurred. The success of this speedy response can be seen in *Figure 7.2* where after each event detection, the system was repaired to a working condition within two data capture intervals after the event occurred.

Figure 7.2 is very similar to the case observed in *Figure 7.3*, however the spike goes in the opposite direction. This is also a common case where a failure occurs on the server and all the debugging and logging software is invoked using up considerable memory while it waits for the user to decide how to handle the failure.

7.2.4 Discussion

Due to the extreme generality of the RAM metric data and the basic scenarios, there are some cases where events occur, but which do not signify any issues with the Grid or any

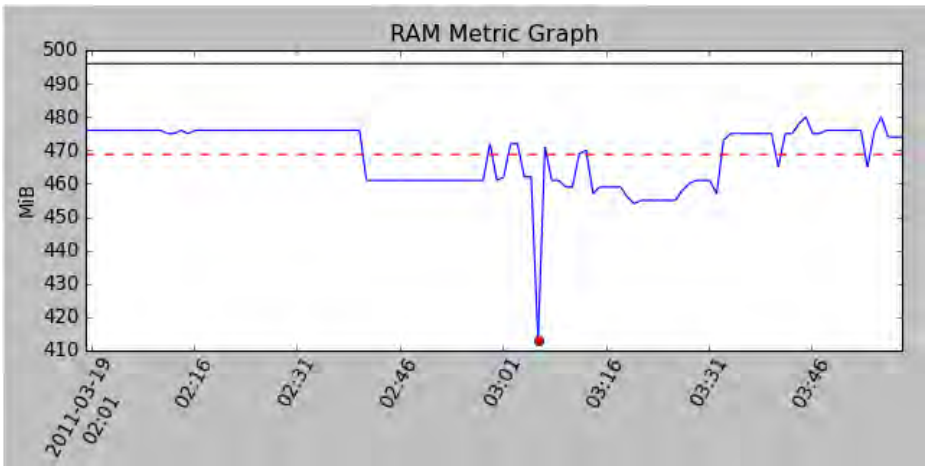

Figure 7.2: RAM metric - sudden dip

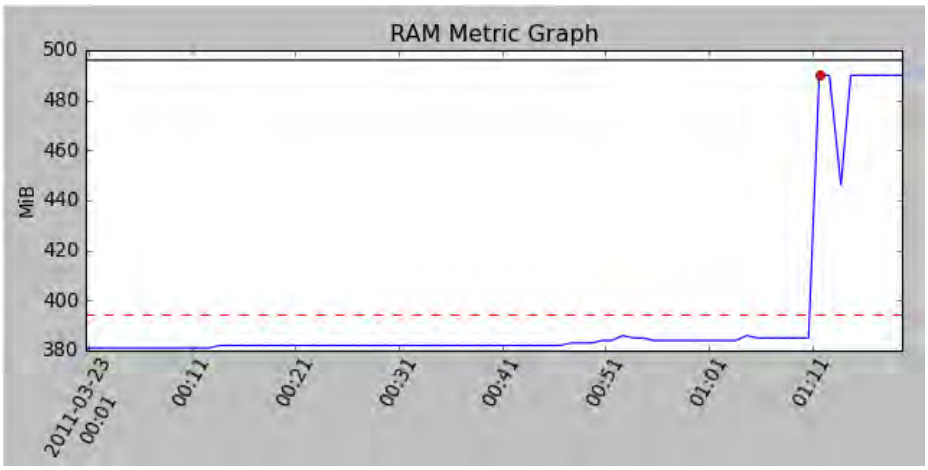

Figure 7.3: RAM metric - sudden spike

services on which it relies. A common explanation for this is that most servers run more than just the Grid services and failure of other services will cause events to occur. Another interesting observation can be seen in *Figure 7.3* where the memory usage climbs suddenly and then maintains the higher level. No explanation can be given for this using only the data from the RAM metric; however, it is most likely either a service or operating system process that is using the remaining memory to optimise performance. This hypothesis could be investigated by implementing a new metric that provides data on which processes are using what proportion of memory.

7.3 Machine Reboot

7.3.1 Scenario

A common scenario when working with production servers is rebooting in order to install patches and upgrades that are vital to the security, stability or feature set of the production server. Since a server reboot causes all running services to stop on power down and the server then attempts to restart them, if configured to do so, upon powering up, it is important to check that the Grid services have indeed been restarted correctly and that all services pertaining to Grid operation are functioning correctly.

7.3.2 Expected Results

Since a machine reboot will unload all processes from memory and as it starts up will have very few services running, it is expected that a machine reboot should be easy for the RAM metric to identify, thereby guaranteeing Grid service availability as soon as the machine has rebooted.

7.3.3 Achieved Results

The RAM metric is very successful in identifying machine reboots as can be seen in *Figure 7.4* where an event has occurred within the first data capture interval after the machine became responsive to incoming connections. However, this rapid identification could result in some side effects as in a few instances an event may be fired before the

operating system has had a chance to start all the Grid services. This is especially noticeable in highly responsive operating systems like Linux and can cause the Grid services to take longer to start since the event may find that some or all of the Grid services are not running and attempt to restart them. This causes a delay as the services are not in a non-running state because they have failed, but simply because the operating system has not yet had a chance to start them.

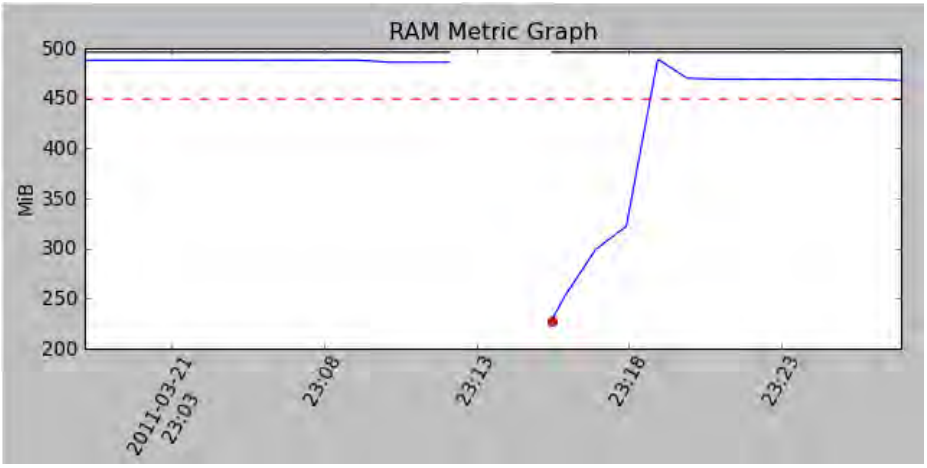

Figure 7.4: RAM metric - machine reboot

Another failure of the metric in this scenario is when the system is consecutively rebooted multiple times. After the first reboot an event is fired to check that the system is working correctly; however, when the system goes down for the second reboot the memory usage is still relatively low. This low memory usage means that there is no real significant change between the memory usage before and after the second reboot. Since there is no significant difference after the second reboot an event will not fire, with the result that the Grid services are not checked to see whether they are functioning correctly after the second reboot.

7.3.4 Discussion

Although this metric is effective in detecting machine reboots, these can also be detected by other means. An easier way to detect them would be to use the built-in operating system log files, which would fix the case where multiple reboots occur in quick succession. However, using built-in log files would be completely operating system dependant and would mean devices with no logging ability would be unable to be monitored.

7.4 Power Outage

7.4.1 Scenario

A less common scenario in this age, with un-interruptible power supplies and backup generators, is a power outage in the data centre that takes down the servers. However, this scenario is much more applicable to Grids where the services are widely distributed across multiple servers.

7.4.2 Expected Results

Since the RAM metric is successful in identifying and responding to machine reboots, it is expected that this success can easily be translated to the essentially unintended machine reboot due to a power outage.

7.4.3 Achieved Results

As can be seen in *Figure 7.5* the system is effective in detecting a power outage and an event occurs within one data capture interval after the machine becomes responsive to incoming connections. These results are very similar to the results obtained for the *Machine Reboot* scenario. This is to be expected as a power outage is essentially the same as a reboot, but with an unsafe machine shutdown and extended machine downtime.

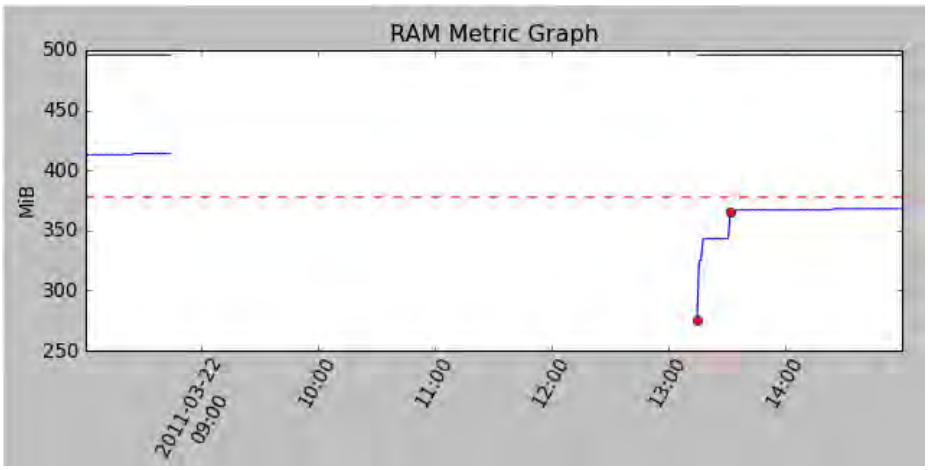

Figure 7.5: RAM metric - power outage

This scenario suffers from the same problem as that in *Section 7.3*, in that it cannot detect multiple power outages in quick succession. However, this is much less likely to occur than multiple machine reboots.

7.4.4 Discussion

This metric is very effective in detecting power outages and can be used to great effect for doing so. An interesting point to note is that the metric cannot distinguish between the lack of data due to the remote machine being unreachable as a result of a power outage at the remote machine or due to infrastructure failure between the host machine and the remote machine. This does not, however, matter as an infrastructure outage most likely implies a Grid service outage as well. The metric could be improved to check that the host machine is not experiencing a hardware or infrastructure failure to reduce the number of possible reasons for the lack of data in the dataset.

7.5 Grid Process Failure

7.5.1 Scenario

A good scenario for testing if the RAM metric is effective in detecting a problem with Grid services is to monitor Grid process failure and observe whether this is detected by the metric.

7.5.2 Expected Results

Since the RAM metric is optimised for detecting when a process crashes by looking for large changes in memory, it should detect when a Grid process crashes. If this process crash is successfully detected, the system should start it up again and return the Grid to a working condition.

7.5.3 Achieved Results

In our extended use of MiG during this study, no Grid process failures were ever detected and no regular use of the Grid caused any fault within the MiG process itself. Thus, in

order to test this scenario we were required to force a Grid process crash. The simplest way to achieve this, was by crashing the Python process executing MiG either by gracefully closing the process or sending the Python process a kill signal thereby terminating the executing Python program.

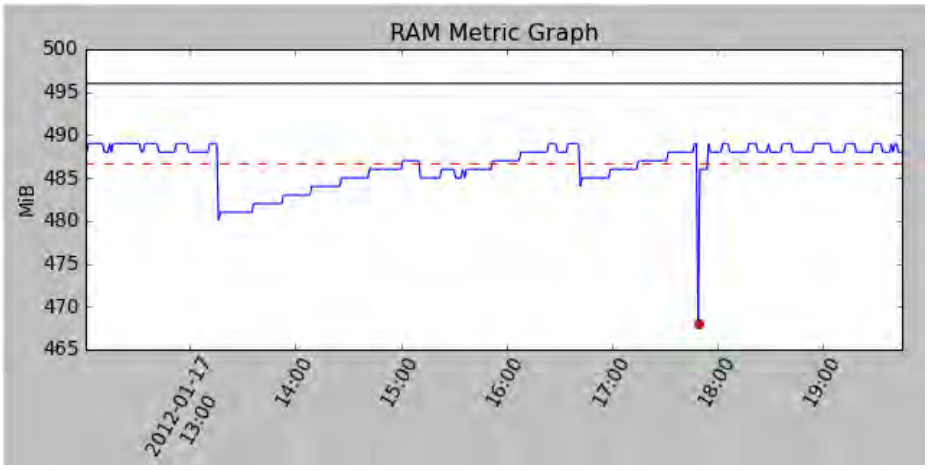

Figure 7.6: RAM metric - grid process crash

Since the Grid process crash was forced, we could easily repeat the experiment multiple times and replicate the results to test the system in this scenario. In doing so it was found that the system detected most of the events, although it struggled to detect events where other running processes were changing their memory use at the same time.

7.5.4 Discussion

Multiple processes concurrently changing their memory use can cause difficulty in detecting event, but this could be avoided by decreasing the polling interval. Descreasing the polling interval will reduce the interval during which the system has no data and the period in which changes in the memory can occur without the system noticing. However, implementing such a change would greatly increase the intrusiveness on the remote system possibly enforcing a re-evaluation of the method used to capture the data. Another option would be to change the way in which the RAM metric works to be based more on processes; instead of monitoring the total memory change it should monitor each individual process' memory usage. This option would be much more intrusive as well as requiring far more in-depth knowledge of how the underlying system and services work. This would also mean that the metric would be specific to each type of Grid and may even need to be changed for each system that it is used with. Another factor to be taken into account is the risk in

the face of a possible security breach. The risk associated with relying on total memory usage is rather low as there is no way an intruder could know if the memory changes are due to his attempts to break in or merely general system usage. On the contrary, the risk using a process based metric is far higher because if an intruder could access this list he would see all processes running on the server and may be able to identify a process with a known or as yet publicly unknown vulnerability and would no longer need to break into the server via the Grid system.

In all other cases the metric was able to detect the forced Grid process crash and to restart the Grid process successfully, thereby reinstating Grid operation. There are a few other edge cases in which the metric had difficulty in detecting the failure of a Grid process, one of which occurred when the Grid process was very small. A small Grid process means that when the process crashes there is only a small memory change, which could easily be attributed to allocation or deallocation of memory by one or more processes. This small memory change is extremely hard to distinguish from general usage small memory changes and thus becomes almost impossible to detect using a memory based metric on its own. Again if a process based metric were used, detection of the crashed process would become easy, however it would again introduce all the problems and risks mentioned above.

7.6 Process Unloaded To Swap Space

7.6.1 Scenario

Another scenario to consider is when data or processes are moved from memory to the swap file or vice versa. This process could cause the metric to trigger an event, even though nothing serious is happening on the server, as there would be a large change in memory usage.

7.6.2 Expected Results

It is expected that the RAM metric should ignore any events where a process has been moved from memory to swap space, or vice versa, as it has been designed with this in mind.

7.6.3 Achieved Results

The RAM metric successfully ignores events where a process is moved from memory to swap space as intended. This is because the RAM metric, as described in *Section 6.3.1*, is designed to account for such movements between main memory and swap space. *Figure 7.7* shows an example of where a process was removed from swap space and loaded back into memory.

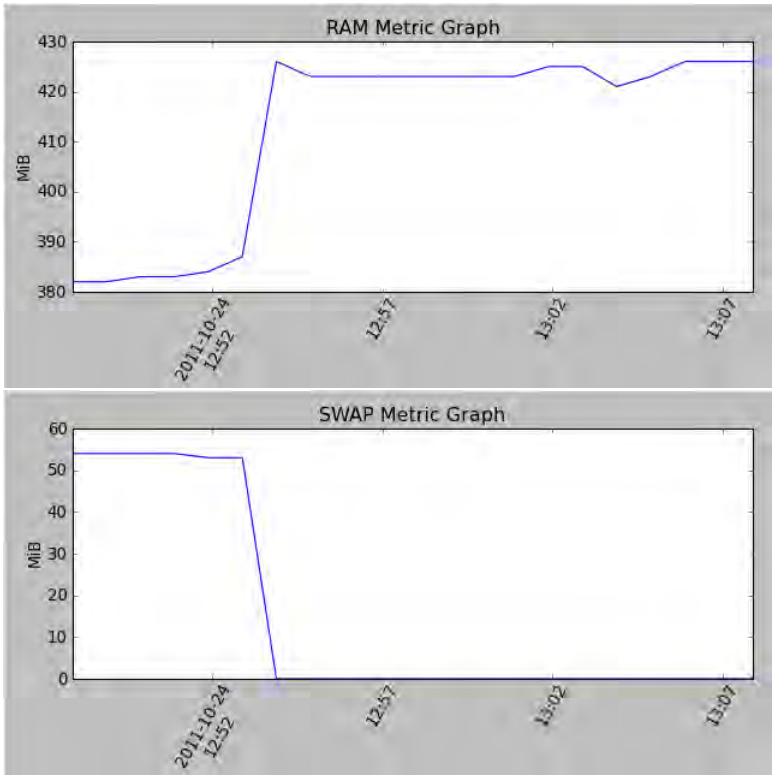

Figure 7.7: RAM metric - process moved from swap to memory

Although not intended, the RAM metric detects memory leaks and application bloating. These occur when an application does not free memory when it closes or when a running application continually requests more memory for use. Memory leaks are quite common in poorly written C programs and application bloat is commonly found in programs written in programming languages that use garbage collection such as Java and Python. *Figure 7.8* shows an example of application bloat occurring in a Python application. From the graph it can be seen that the application requests more memory and pushes some of its paged memory into the swap space.

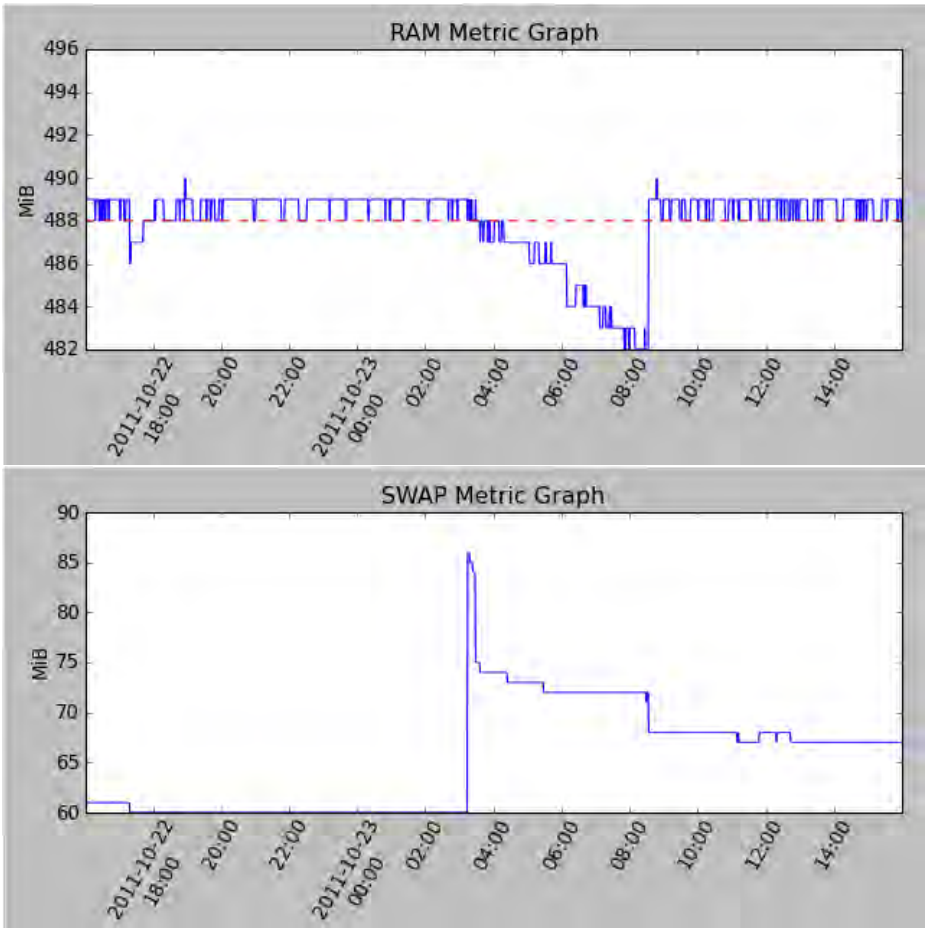

Figure 7.8: RAM metric - bloat detection

7.6.4 Discussion

It is not surprising that events occurring while processes are moved to swap space are successfully ignored, as the metric was designed to deal with this. However, the detection of memory leaks and application bloating is unexpected, albeit a very useful addition for debugging production servers. This additional ability of the metric allows administrators to observe the server and see if there are any services running that are leaking memory or bloating as well as providing the time at which the leak or bloat occurred. The important point to note here is that the detection provides a time stamp for administrators, thereby assisting in debugging when the error occurred in scheduled scripts that are being executed.

Due to the use of swap pre-fetch, whereby the application data is kept in the swap space to allow a quick swap into and out of memory without needing to write to the disk, the results for application bloating could be misinterpreted. Application bloat may be misdiagnosed due to the high amount of data being kept in the swap space for the purposes of pre-fetching. However, it should still be noticeable when the amount of data in the swap space increases continuously over a period of time. This slow increase in swap space is incredibly hard to detect in a short time-frame and may only be detectable by observing the swap space usage over a period of days or even weeks.

7.7 Summary

The RAM metric provides very useful information for development and production based servers. It can effectively predict Grid system crashes and successfully recover full Grid operation after the crash has been detected by correcting any errors found. This effective recovery method allows the Grid to be fully operational at all times, even when system administrators are not monitoring the Grid. If the Grid cannot be repaired by the expert system, the system administrators can be alerted via a medium of their choosing to the fact that the Grid is not functioning correctly.

Chapter 8

Grid Jobs Metric Results

This chapter presents and describes the experiments conducted on the grid jobs metric, as described in *Section 6.3.2*. It then goes on to present the results of these experiments and a discussion thereof.

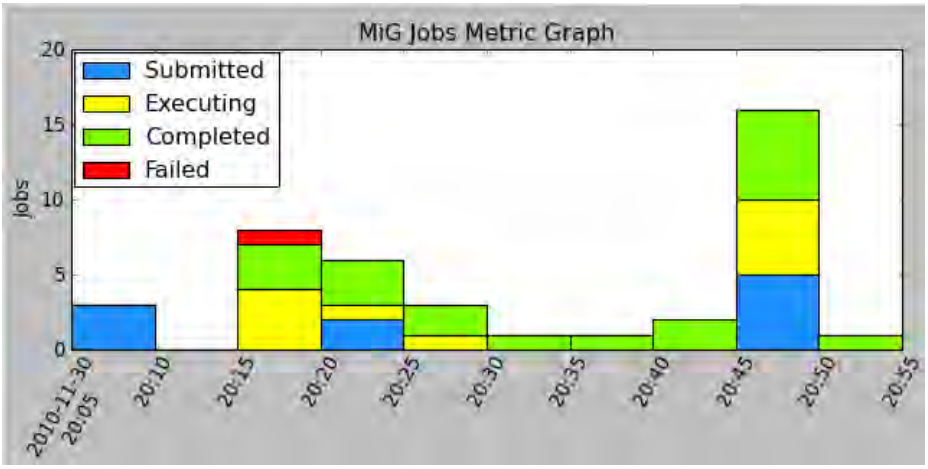

Figure 8.1: Example graph of grid jobs metric

Some general capture data can be seen in *Figure 8.1*, where various jobs have been submitted and processed, and returned by the grid. We would expect to see this type of graph if all services are working correctly on the grid server and no problems have occurred within the given time-frame. This graph is exceptionally useful for grid administrators to gain a general understanding of the execution state and load on their grid.

To be able to interpret the graph shown in *Figure 8.1* for the grid jobs metric it is important to know what each of the bars depicted in the various graphs means. The blue bars show

the number of job submitted to the grid during the interval. The yellow bars show when jobs have been fetched off the grid by a resource for processing, while the green bars show when a resource returns a job that is has finished processing to the grid, together with all the job results and data files. Red bars are only shown when a job fails to be executed on the resource. This commonly occurs when either the resource signals the grid that execution has failed, or when the grid halts execution because the job has either exceeded its execution time limit, or the resource has become unresponsive and the grid is no longer able to determine the status of the resource or job being executed once the resource time-out has been exceeded.

For the purposes of testing the grid jobs metric, old logs were obtained from the MiG project's production server as used by them for a high performance computing course that they conducted during 2011. These logs were parsed and loaded into the repair system's database for analysis and testing. Unfortunately since the logs only contained data relevant to the MiG server process itself, the repair system could not be fully tested using only this data. However, the data did provide a better opportunity for data mining than data generated specifically for this research by creating and running jobs on the server itself. Self generated data will always have less of an academic use that real data used in an academic context. Thus the real data was used to test the detection algorithms and scenarios were generated in order to test the ability of the repair system to detect and repair real faults.

8.1 Job Event Activity Spike

8.1.1 Scenario

One of the most basic scenarios that should be intercepted by the grid jobs metric is a sudden large change in the grid jobs activity. As described in *Section 6.3.2*, this is the basic use case for the grid jobs metric and detects large activity spikes.

8.1.2 Expected Results

Since the grid jobs metric uses a statistical approach for detection, it is expected that the metric should perform fairly well. The metric should not make detections when the data is fairly consistent, but should make detections when the data is changing rapidly.

8.1.3 Achieved Results

The grid jobs metric performed as expected, triggering an event when the data fell outside the given statistical window. *Figure 8.2* shows a graph of the system detecting events when the data profile changes.

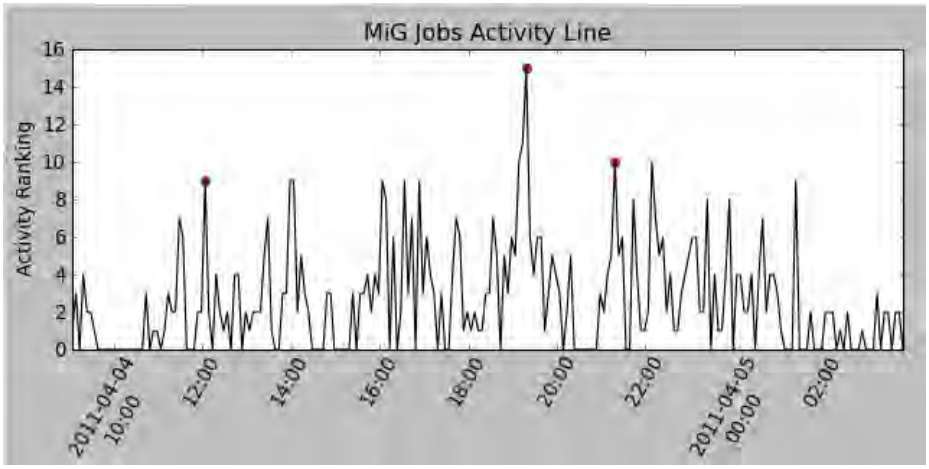

Figure 8.2: Grid jobs - activity spikes

From the graph in *Figure 8.2*, it can be seen that the grid jobs metric successfully identifies points at which the number of jobs falls outside of the recently known pattern for job events. At these identified points the grid can be checked for security breaches and repaired if necessary; otherwise a system administrator is alerted to the security breach or fault.

8.1.4 Discussion

The ability to detect large spikes or dips in the job event data is quite important. A large dip in the job event data can be a very good indicator that a fault has occurred on the grid and that the grid server has lost contact with a client and resource base or is unable to continue receiving and processing jobs. An early indicator of a failure is vital in any attempt to keep a grid service active and processing data in real-time, as it allows diagnostic and repair processes to begin before the rest of the grid has noticed the failure. Thus, the rest of the grid may never notice that any failure occurred since the early indication of the failure allows the grid to be repaired almost immediately after the fault has occurred. It is also important to detect large spikes in the job event data as this may be a simple indicator of an unwanted intruder in the grid system or that one of the grid clients is spamming the grid system with job requests. Another possibility is that a

resource is requesting and processing jobs but failing to process them properly and thus returning an incorrect result to the grid. All of these events are problems that require immediate attention with a fast reaction as they could quickly bring grid productivity down to zero. At the very least, the grid should be inspected to confirm that it is operating correctly, and if an unknown fault is found the system administrator should be informed.

Figure 8.2 shows three points where events have been detected due to irregular job activity detected on the grid system. The first detected point at 12:00 is where the system has detected irregularity, which is most likely due to the period of zero activity around 10:00 that lowered the average activity level of the region. The second detection around 19:00 is due to the job activity at that point spiking to almost double the local average for job activity. The third and final detection point, shown at around 21:00, is again a detection most likely due to the period of low and zero activity just prior to it, which skews the activity average towards zero and thus causes this high spike to be detected.

8.2 Job Event Inactivity

8.2.1 Scenario

A scenario that is very useful in identifying whether there is something wrong with grid operation is detecting when the grid experiences a prolonged period of inactivity. This inactivity could be due to multiple factors, such as a lack of jobs to process, no suitable resources available that can process any of the jobs in the queue, or a problem with grid operation or the grid process itself. The first two problems are not correctable by any means and are a standard part of operating a grid. The final problem is correctable and may be the reason for the prolonged inactivity. If a grid fault is the reason for the prolonged inactivity, it is vital that the grid is repaired and allowed to resume normal operation so that jobs can once again be processed.

8.2.2 Expected Results

Given that there are jobs running on the grid, it is expected that detection of inactivity should be fairly accurate. However, it is also expected that this detection of inactivity cannot be instantaneous. This stems from the fact that it is impossible to tell if the inactivity is due to a momentary dip in grid activity as a result of normal grid operation

or a fault. Thus a significant period of inactivity is required before it can be decisively assumed that there is a grid fault causing the inactivity.

8.2.3 Achieved Results

Figure 8.3 shows a period of inactivity on the grid jobs metric, which is due to normal grid operations. The graph also shows that after a short period of inactivity the metric successfully detects this inactivity period. After making this detection the metric hands over the case to the expert system for analysis and, if necessary, repair. In this case, the expert system did not need to perform any operation to repair the grid and simply marked the inactivity as being the result of normal operation since it coincided with a common period of inactivity during early morning hours throughout the dataset.

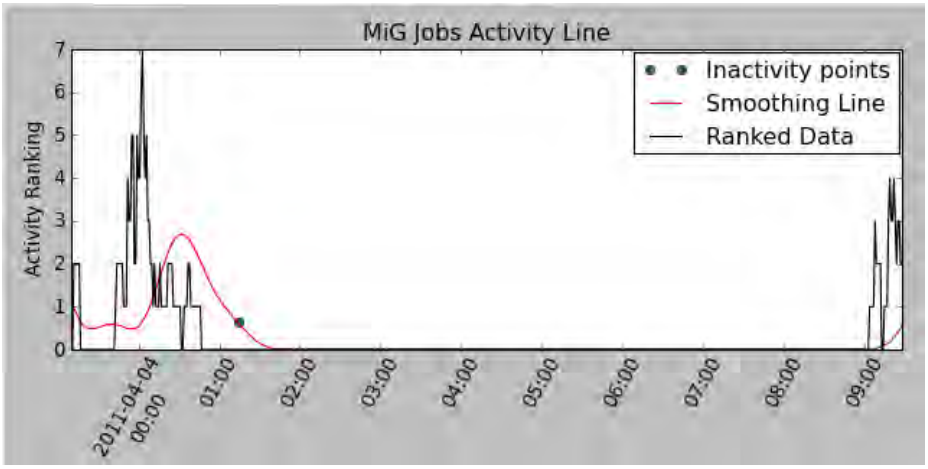

Figure 8.3: Grid jobs - inactivity detection

From *Figure 8.3*, it can be seen that the smoothed line, shown in red, works fairly well in detecting job inactivity points, shown as green dots. This detection occurs approximately thirty minutes after the last job activity occurred.

8.2.4 Discussion

Detecting a grid fault through the observation of grid inactivity is a non-trivial task. As stated above the inactivity shown in *Figure 8.3* is due to no fault of the grid and the grid is operating normally during this time period. The fact that this inactivity is due to normal grid operations could be deduced from the shown time period of the inactivity, however this may not be the case for Grids that are used across multiple time zones and continents.

Detection through the observation of inactivity is by definition flawed in that it is impossible to determine indubitability when inactivity has occurred due to a grid fault or simply no activity being present on the grid. Any approach that is used will never be completely effective and will only give an indication that there may be a fault. Although not effective it is still important to observe the grid inactivity as this is a good indicator of when a fault on the grid has occurred. The cause of this fault could stem from a number of areas and may not simply be a case of grid processes failing on the grid server. There are numerous other reasons why jobs may not be submitted or executed, including no resources being currently active to process jobs, no clients submitting jobs that can be executed by the current resources available, the grid network being unreachable due to a fault or failure by clients and resources, or simply that there are no clients wishing to submit jobs for execution. Given that there can be several reasons for any period of inactivity on the grid, detecting inactivity on the grid clearly does not mean there is a fault on the grid and therefore, this cannot be assumed when inactivity is detected. However, when a long period of inactivity is detected it is always best to check that the grid is operating correctly. Although not implemented, a way of reducing the number of times the grid needs to be checked due to inactivity could be reduced by pattern matching to past data; i.e., checking when inactivity has occurred in the past and whether these periods match the current period or attempting to match previous usage activity patterns, e.g., the grid may repeatedly go through a predictable pattern of a period of usage followed by a similar period of inactivity.

8.3 Job Failure Monitoring

8.3.1 Scenario

An interesting detail to monitor with respect to grid systems, is that of jobs returned as failed or unable to complete. The cause of such failed jobs may be one of the following: failure to execute; failure by the resource to process the job as the job requires something the resource does not have; failure by the resource to return a result or return the job as completed; or failure by the resource to complete the job due to the maximum execution time having being exceeded.

8.3.2 Expected Results

The grid jobs metric is expected to detect all instances of failed jobs. However, the effectiveness of the failure detection depends on the detection limit set to process a resource as a malfunctioning resource.

8.3.3 Achieved Results

Figure 8.4 shows all the failed job data over the period. The graph shows a large spike on job failure events that occurred at around 14:00 and were quickly detected by the repair system.

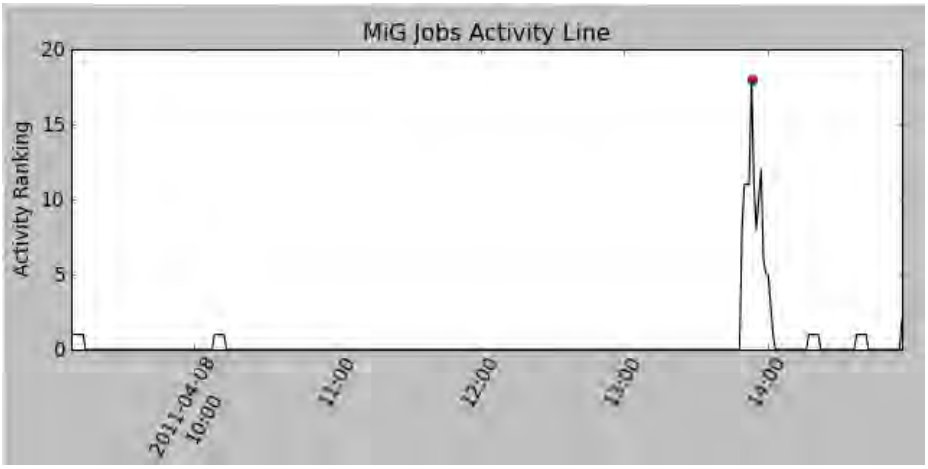

Figure 8.4: Grid jobs - job execution failure detection

8.3.4 Discussion

Although this is not useful in terms of fixing a fault on the grid, observation of these failed jobs could significantly improve the quality of the grid. Resources that return an unreasonable number of failed jobs in a short period of time could be removed from the queue for receiving new jobs and the resource administrator notified. Once the resource administrator has repaired the resource node and confirmed that it is working correctly, the resource could be reintroduced onto the grid for job execution. This should reduce the number of job failures occurring on the grid and means that jobs that execute have a higher chance of completing successfully.

8.4 Resource Request Monitoring

8.4.1 Scenario

Another interesting aspect that can be monitored on the grid is the requests for jobs by resources. A resource requests a new job when it has either completed its current job or recently come online and is requesting a new job for the first time during the current session. When a resource requests a new job, it is either allocated a user submitted job to complete or a time wait job that makes the resource wait for a five minute period before once again requesting a new job from the grid server.

8.4.2 Expected Results

Since the request for jobs is pretty consistent, it is expected that the grid jobs metric should be quite successful in detecting when resources have failed or been disconnected from the grid.

8.4.3 Achieved Results

Although detection of resources dropping off the grid is not instantaneous due to the five minute waiting period between requests, the grid jobs metric can successfully detect that a resource has left the grid system within fifteen minutes of the resource leaving.

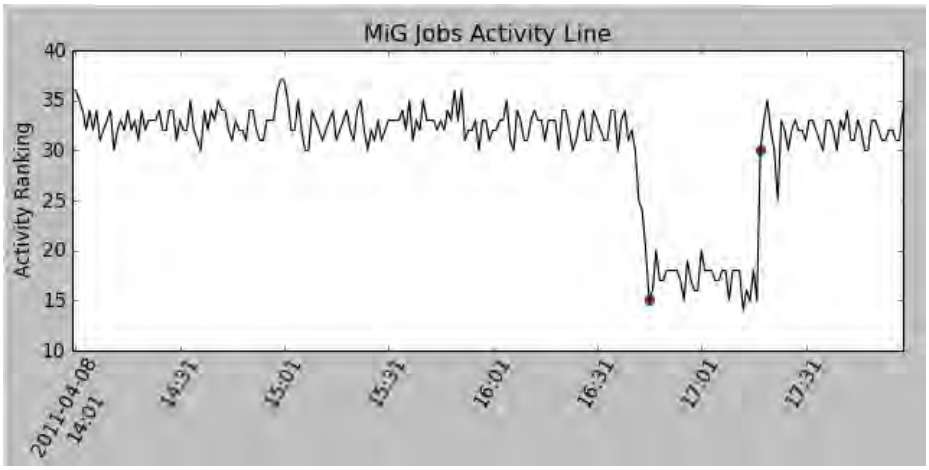

Figure 8.5: Grid jobs - resource stops requesting jobs

However, the grid jobs metric cannot easily detect that a resource, currently computing the result of a job, has left the grid system as it assumes the resource is still busy with the computation. Until such a time as the resource reconnects and requests a new job, or the job execution time limit passes, it is not known whether the resource will return a result, be it a failed or completed result, or request a new job. If the resource fails to do either of these, it is known that the resource has left the system. The graph in *Figure 8.5* illustrates the scenario where a resource has stopped requesting jobs for a period, and shows that the system detects both when it leaves and again when it reconnects.

8.4.4 Discussion

Detecting when a resource has left the grid system is very easy to do under normal circumstances as the resource will simply return the result of the last job it was given and not request a new job. However, there is a delay in the detection of this due to the possibility of a delayed connection from the resource. De-synchronization between the grid system and resource can occur easily and can be explained by a number of reasons, for example: there was a delay in starting the waiting job on the resource; the resource had to queue to connect to the grid server; or the resource's wait job was interrupted and had to be restarted. All of these reasons are logical explanations as to why the resource may not re-connect with a completed result job at the time it was expected to, and hence there is the need to wait for at least two intervals before assuming that the resource has disconnected from the grid and will not be returning a result.

8.5 Summary

The grid jobs metric is fairly successful in detecting apparent problems as they occur on the grid by monitoring the grid scheduler for jobs submitted, executed, completed, and failed. This means that any unexpected or high deviations in jobs submitted, executed, completed, or failed will cause a repair event to be triggered. This repair event will then diagnose the grid system, checking for any faults that may have occurred to explain these phenomena.

Chapter 9

Evaluation of Results

This chapter analyses and comments on the results shown in the previous two results chapters, *Chapters 7* and *8*.

9.1 Introduction

To evaluate the results in a meaningful manner, an evaluation technique and criteria first need to be established. The criteria that make the most sense for comparing the repair system results are: performance, accuracy, and effectiveness. Since these criteria are vastly different from each other, the evaluation will be divided into three categories each related to a single criterion. Thus, each category will have its own requirements and metric by which the compliance of the results will be determined.

9.2 Performance

One of the key factors when working with a real-time system is the system's maximum throughput capability. The current system implementation is single threaded and therefore limits the maximum number of monitored systems to that which can fit within a single cycle of the server thread, which is essentially the length of the polling interval. This single threaded system worked well for the fairly small number, less than twenty, of systems monitored to produce the results shown in this thesis. However, the system can easily be adapted to perform the monitoring requests in a multi-threaded mode; spawning a

new thread for each system to be monitored and performing any data requests inside this thread. This multi-threaded mode would dramatically increase the maximum throughput capability as it would no longer be capped by the fixed length of the polling interval and would allow the system administrator to set a much lower polling interval if required. Presently, the minimum polling interval is simply the longest time a monitored machine takes to process a monitoring request and return the relevant data. However, using a multi-threaded approach with a short polling interval would create a new bottleneck at the data storage point in the database, where all the data from the monitored machines is stored. If the database cannot store all the incoming data from a single polling cycle before the next cycle starts, a backlog would occur in the database execution queue.

Backlogging the database execution queue would mean that eventually the input buffer for the database would become full and as a result, it would randomly drop requests. Since the database is now the bottleneck, certain database specific actions could be used to increase the insert speed of the database [35]. Performing testing on MySQL using bulk inserts resulted in a value of 4851 rows inserted per second, however this value depends on the hardware specifications of the machine on which the repair system is run. This means that if a metric only inserts one row per machine logged, the following equation holds true where n is the number of machines monitored,m is the number of metrics per machine, and *pollint* is the polling interval:

$$\frac{n * m}{pollint} = rows/sec \tag{9.1}$$

Therefore, the maximum number of machines that can be monitored can be calculated as follows, assuming the polling interval is 60 sec and the number of metrics in use is three:

$$\frac{60sec * 4851rows/sec}{3metrics} = 97020 \text{ machines} \tag{9.2}$$

Thus, the current system can support up to 97020 machines being monitored before an additional database server would be required to process all the incoming transaction requests.

Another factor to consider with respect to the system's performance is the response time of the system. The response time of the system is measured as the elapsed time from when an incident occurs to the time when the system detects that the event has occurred and reacts to it. For the time being, it is assumed that the system always detects an event at some point after it has occurred; a discussion of where this is not the case is presented in *Section 9.3*. Firstly, it should be noted that the system cannot be expected

to respond any faster than the polling interval since, if the error occurs immediately after the last poll, the next poll would be the first opportunity for the system to obtain data after the error event has occurred. Thus, it is important to have a relatively short polling interval that allows the system to react to events quickly after they occur. The default polling interval is specified as one minute, however for time critical services this could be shortened. However, care must be taken when shortening the polling interval so that the system is not overloaded, as described above. A separate repair system could be used to monitor time critical services with a significantly shortened polling interval. As long as the separate repair system also has separate dependency services it should not affect the original repair system. However, to ensure that there is no interference the repair systems should be run on separate hardware.

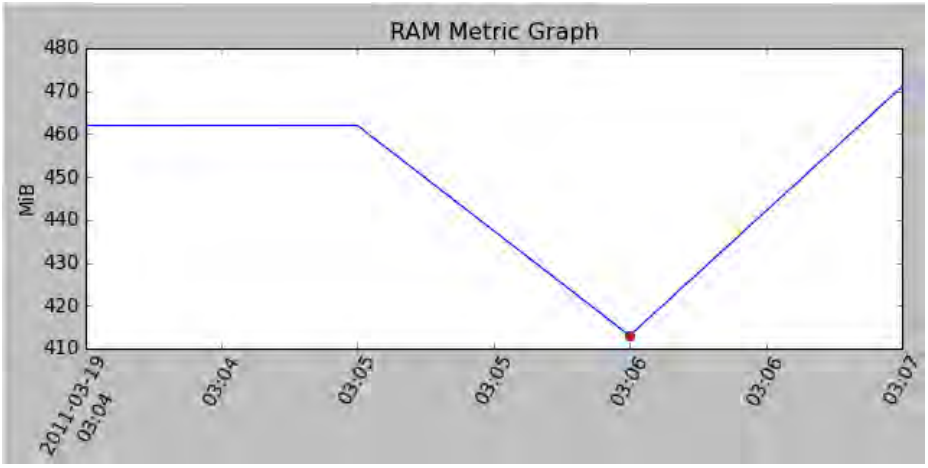

Figure 9.1: RAM metric detection speed with 60 second polling interval

Figure 9.1 shows a short section of the RAM metric data where an event has been detected. This event is known to be a restart of the repair system front-end that provides system administrators with graphs and statistics. The figure shows a 3 min interval with 1 min polling. Therefore, there are only four data points used to generate this graph and it can be seen that as soon as the data arrives from the second last poll, the fault is detected. Moreover, by the time the last poll returns data, the repair system front-end has restarted and the memory usage is once again at the expected level. This clearly shows how quickly detection of events occurs, and that even the restart of a process that takes as little as 20 sec can be detected. Using statistics to calculate the probability of a process restart being detected given that the restart takes approximately 20 sec can be done as follows:

$$\text{probability} = \frac{\text{restart time}}{\text{polling period}} \tag{9.3}$$

Using this equation it can be calculated that a 20 sec process restart has a 33% chance of being detected using the default polling interval of 1 min. If the polling interval is reduced to 30 sec, the percent chance of detecting the process restart doubles to 66%. Unfortunately, in order to guarantee detection of any process restart when using a polling system, the polling interval must be smaller than the shortest process restart time. However, since some processes such as Apache have a very short restart time, often in the region of 1 sec, it is impractical to attempt detection of all process restarts.

9.3 Accuracy

Accuracy of the repair system is a very important factor in evaluating the usefulness of the repair system. A repair system that does not detect fault events immediately, or shortly after they occur, is of no use as it will not repair the fault in question in a timely fashion. Thus, the grid system and services will be affected with extended downtime or limited functionality until either the repair system detects what it considers to be an event and runs a diagnostic at some point in the future, or the system administrator becomes aware of the problem. Similarly, a repair system that continuously detects faults when none have occurred will burden the grid with unnecessary diagnostic tasks, only to discover that no fault had in fact occurred. Thus, it is important that the repair system detect when faults occur with a slight detection skew towards false positives, so that the grid remains functional with as little diagnostic overhead as possible.

Due to the difficulty of diagnosing events that occurred in the past, it is very difficult to come up with exact figures of how many real events occurred, how many of these events were detected, and how many of the events detected were not real problems. This is especially true of the data used for the grid jobs metric as no access was available to the grid machine and only the grid system logs were supplied. Thus, to arrive at a conclusion of how accurate the system is, we performed a number of simulated tests to determine the accuracy of the metrics. These simulated tests were designed to provide a good understanding of how accurate the metrics are and under what circumstances they become inaccurate and start missing event detections.

The RAM metric, as described in *Section 6.3.1*, performed fairly well under accuracy testing and detected the majority of events that occurred. However, due to the way the RAM metric detection algorithm was designed, using percentage based detection for events, the RAM metric fails to detect process crashes where the process uses only a small

amount of memory during its lifetime. Although the percentage based detection performs well, the percentage used would need to be tweaked per monitored machine depending on the size of the smallest process to be detected and the value the percentage is based upon. For instance, if the percentage were based on the maximum amount of memory available to the server when the server average memory consumption uses less than half of this available memory would not facilitate producing adequate detection. An alternative to this would be to use fixed value detection and this would improve detection of small processes. However, on large servers running many services this fixed value detection could cause event detections during normal operation simply due to the minor increases and decreases in memory usage of each running service. In this case, percentage based detection methods are far more useful and given that a reasonably large period of data exists for the monitored machine, a good basis for percentage based detection would be the mean memory consumption over the data period with outliers further than two standard deviations removed. By removing outliers in this way, it is far more likely that the true expected mean of the memory usage will be obtained.

Tables 9.1 and *9.2* give the results of the RAM metric's accuracy being tested on a Debian x86 and a Fedora Core server. As suggested above, the percentage used for detection in the RAM metric needs to be tweaked on a per server basis to detect the grid process consistently. However, with prior knowledge of the normal memory usage of the server, the correct value for consistent grid process detection can easily be calculated. This can be done using the following equation, where s is the size of the process to detect, m is the normal memory usage of the server, and p is the percentage used for detection:

$$\frac{s}{m} \geq p \tag{9.4}$$

Unfortunately this formula only works perfectly if no interference can be assumed from other processes, and thus, the percentage needs to be reduced slightly below the value obtained from the equation to guarantee that the required amount of memory change will be detected. A suggested solution is to decrease the percentage obtained by 10%:

$$\frac{s}{m} * 0.9 = p \tag{9.5}$$

The grid jobs metric, as described in *Section 6.3.2*, also performed fairly well under accuracy testing and detected the majority of events that occurred. Due to the grid jobs metric detection being heavily based on statistics, the detection suffers from severe falloff

Table 9.1: RAM Metric Accuracy Test on Debian x86 with an average memory usage of 380 MB

Test Conducted[1]	Detection %[2]	No. of Tests	Accuracy
Repair System Front-end Crash (80 MB)	10%	10	100%
	5%	10	100%
	2.5%	10	100%
Grid Process Crash (10 MB)[3]	5%	10	0%
	2.5%	10	80%
	2%	10	100%
Grid Process Crash (30 MB)[4]	10%	10	30%
	5%	10	100%
	2.5%	10	100%

[1] Supplied process sizes are approximations.
[2] The detection % is the percentage change used to perform detections in the RAM metric algorithm.
[3] This is an empty grid process with no jobs in the execution queue.
[4] This is a loaded grid process with around 250 jobs in the execution queue.

Table 9.2: RAM Metric Accuracy Test on Fedora x86 with an average memory usage of 490 MB

Test Conducted[1]	Detection %[2]	No. of Tests	Accuracy
Repair System Front-end Crash (80 MB)	10%	10	100%
	5%	10	100%
	2.5%	10	100%
Grid Process Crash (10 MB)[3]	5%	10	0%
	2.5%	10	50%
	2.0%	10	80%
	1.8%	10	100%
Grid Process Crash (30 MB)[4]	10%	10	0%
	5%	10	90%
	2.5%	10	100%

[1] Supplied process sizes are approximations.
[2] The detection % is the percentage change used to perform detections in the RAM metric algorithm.
[3] This is an empty grid process with no jobs in the execution queue.
[4] This is a loaded grid process with around 250 jobs in the execution queue.

after a certain number of data spikes and dips have occurred. If a theoretical data set is generated with the data alternating from zero to 100, and the detection method tested against this data set, the severe falloff is clearly apparent in the resulting graph. *Figure 9.2* illustrates this in a 24 h period where detection falloff occurs after 395 events have been detected. It is at this falloff point that data spikes and dips are no longer considered events as they have become accepted as normal data by the statistical detection methods used in the grid jobs metric.

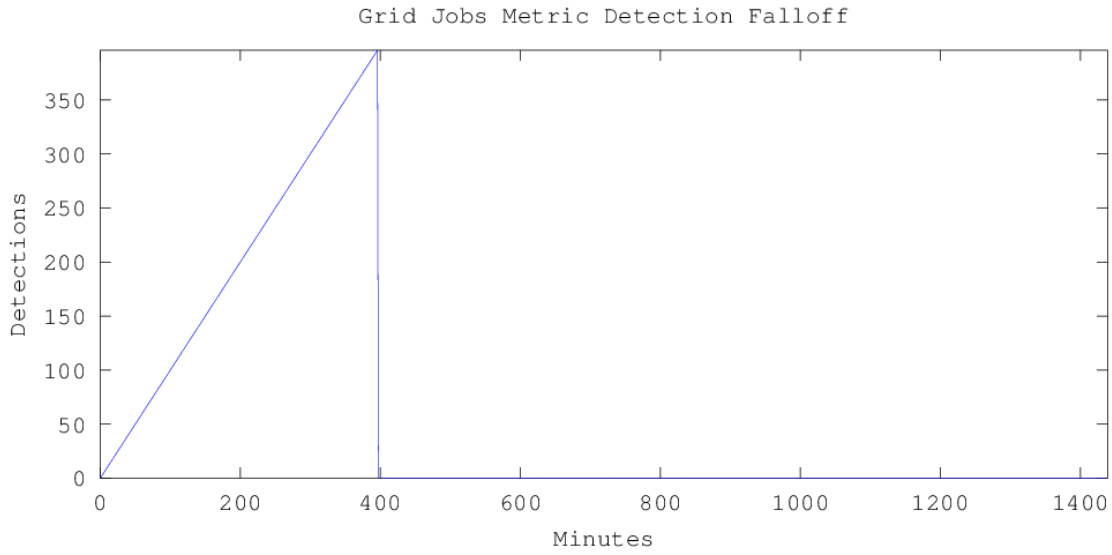

Figure 9.2: Grid jobs detection falloff after 395 events

Other than the theoretical falloff problem, the grid jobs metric is accurate within its design specifications. Points in the data where there was a sudden change in the trend of the data were correctly detected as well as all events that fell outside of the statistical detection interval. Moreover, there were not a huge number of false detections, as almost all the points when taken in context, could have signified possible problems.

9.4 Effectiveness

The ultimate question for a repair system of this sort is, does the repair system have a positive effect on the grid system and can the repair system perform as well as or better than a system administrator? To determine this we need to test the response time of the repair system with respect to detection of events and its effectiveness at effecting repairs. Thus, to do this in an unbiased manner, in this section it is assumed that the repair system

can detect all actual events consistently and events that the repair system is not trained to repair will not be included.

We have already shown in *Section 9.2* how quickly the repair system can detect events that occur. However, this rapid detection does not answer the question of how effective the repair system is at correcting the detected error and allowing normal operation to resume on the grid system.

Table 9.3: RAM Metric Effectiveness Test on Fedora x86 with an average memory usage of 490 MB

Test conducted[1]	No. of Tests	Detection Time[2]	Repair Time[2]
Repair System Front-end Crash (80 MB)	10	32.91 s	0.5 s
Grid Process Crash (30 MB)[3]	10	29.74 s	0.8 s
Grid Jobs Inactivity	10	31 min	0.9 s

[1] Supplied process sizes are approximations.
[2] Times listed are the averages over all the tests conducted.
[3] This is a loaded grid process with around 250 jobs in the execution queue.

Table 9.3 shows the average detection and repair times for the repair system to detect and repair the listed events. Although the detection time is not amazing, it is what was expected since the detection time is strongly related to the polling interval. Thus, if the occurrence of events is evenly distributed along the polling intervals then the expected average time to detection will be half the polling interval. The repair time is a little more unpredictable than the detection time as there is no factor on which the time taken to repair hinges. The factors that affect repair time are all to do with what occurs when the expert system is started for grid diagnostic analysis and repair. Each test that the expert system has to run to determine the root cause of the fault, or whether a problem has in fact occurred, will take a varying amount of time. Thus, it is the number of tests that are run that will ultimately determine how much time is taken to perform the diagnosis, and it is impossible to determine the number of tests that will need to be run without prior knowledge of what the underlying cause of the fault is.

Another factor that affects repair time is how many execution attempts are made of a single test before it can be decided that the test has failed. An example of this would be to ping the remote server: how many ping attempts should be made before it is decided that the remote machine is unreachable or not replying to ping requests? With

the current implementation of the system, the expert system only executes each test once per instantiation. However, the expert system could be configured to execute four ping tests per instantiation for instance and accept a single ping reply as a successful test. Nevertheless, a multiple ping test such as this would be much better suited to testing for intermittent connectivity, and thus reporting a possible line fault to the system administrator, as only 50% of 40 ping replies were actually received.

A special note must be made of the detection time for detecting inactivity on the grid jobs. This exceptionally high time is due to the algorithm used to determine when the grid is undergoing a period of inactivity, as described in *Section 6.3.2*. After the last job event has occurred, the algorithm takes time to understand and identify that no more job events are occurring within a reasonable period of time. Only after this period of time, which is dependant on the recent job event data, has elapsed, will the algorithm flag the grid as inactive and request the expert system to conduct a diagnosis to check that the grid is in a fully operational state.

Chapter 10

System comparison

This chapter presents and compares the differences between the system as described in this thesis and readily available products designed to achieve roughly the same tasks. It then goes on to discuss and comment on these differences and the implications thereof.

10.1 Big Brother

10.1.1 Description

Big Brother is a web-based software tool most commonly used by system administrators to monitor computer systems and networks. Big Brother's primary role is to make sure that machines and systems stay up and running. Big Brother uses a client-server based model with its own protocol, whereby the clients send status updates and information in a relatively secure fashion over port 1984 to the server. Big Brother provides a web-based graphical interface that is available for anyone to see by default and shows the health status of each server or service as either green or red. The display of green for an item denotes that the server or service is in good working condition and operating normally; whereas the display of red indicates there is a problem with the system or one of the subsystems it relies upon. This health status is determined by performing simple tests on a system or machine, such as testing output or checking that a process is running. A problem with the system or machine is identified if either the incorrect output is produced or it fails to respond in a reasonable amount of time. [24, 25]

10.1.2 Comparison of Features

10.1.2.1 Monitoring

Big Brother has the ability to monitor the connectivity and operability of a multitude of services remotely over the network by connecting to the service port, requesting data from the service, and checking that the service provides the correct output. These services include the following: HTTP, POP3, SMTP, FTP, NNTP, and DNS [7]. In addition to monitoring services by requesting data, Big Brother can also ping a machine to determine the state of connectivity to the machine. The client program that runs on the remote machine reports to the server every five minutes returning the available disk space, the five minute CPU load, any notices and warnings, and a list of the running processes. The results of these scripts are then returned to the monitoring server via the network using the Big Brother protocol. [24, 25]

In comparison, our proposed system has the ability to monitor network ports remotely; however, it does not perform such monitoring at regulated intervals. The monitoring of connectivity on ports is performed when the system is attempting to diagnose and repair a possible fault that has been detected. The system also attempts to ping the target machine if no response is obtained on a service port during fault diagnostics to determine if the entire machine is unreachable or just the specific service. The proposed system differs significantly in that it does not require any specific client-side software other than an SSH server to which it can connect to run diagnostic scripts. SSH servers are installed by default on most Linux and Unix platforms, and are fairly simple to install on Windows server platforms. Thus, instead of having the client program run every five minutes, the system can be dynamically configured with a specific interval at which to poll each remote machine it is currently monitoring. Instead of the Big Brother protocol, the proposed system sends all console output from the diagnostic scripts back to the server across the encrypted SSH connection. This output includes any errors that may have occurred while running the diagnostic scripts, thus allowing the system to gain a better understanding of any problems on the remote machine and correct any dependency issues that may occur when connecting a new machine for monitoring to the system.

10.1.2.2 Alerting

One of Big Brother's main features is the alerting feature that allows Big Brother to alert the system administrators if there is a problem on one of the services or servers that Big

Brother has been configured to monitor. The main method for doing this is via email, although Big Brother can also be configured to use pagers and SMS notifiers via either the depreciated or third party plug-ins. Big Brother also provides a web-based interface whereby the system administrators can quickly determine which systems have failed tests or issued warnings.

In contrast to this, the system proposed in this thesis has no real alerting features built-in by default as this is not what it was designed for. Instead the system attempts to repair any problem detected and only if this repair fails, is the administrator notified of the event that caused the system to run the diagnostics, including the diagnostic commands that were run on the remote machine together with their output. This gives the system administrator a significant advantage in that basic diagnostic output is provided, instead of him having to start from the beginning and proceed to performing advanced or service specific diagnostics to locate the potential fault. Although not integrated in the actual monitoring system, the graphs produced by the proposed system as shown in this thesis, can be displayed to system administrators allowing them to gain a better understanding of what is happening on the monitored systems.

10.1.3 Discussion

Big Brother is primarily a tool for system administrators to identify systems and services on the network that are not functioning correctly. However, Big Brother does not have any in-depth knowledge of how these systems work or how they are integrated. As such, this kind of system is suitable for a regular production environment where the failure of one service only affects a few systems or users. However, for a Grid system, where normal operation of the system is dependant on many systems and services working together at the same time, it can show that a system or service is not functioning, but cannot identify all the other systems or Grid operations that are no longer working on the network as a result of this single failure.

In addition, Big Brother has no mechanism for repairing any faults that occur on the network; in fact, Big Brother has no mechanism for sending any kind of command to a remote machine as the remote machines only report to the Big Brother server and do not accept commands. This means that even for simple problems such as the restart of a process on a server, the system administrator has to get involved and perform the repair operation. This is a huge drain on the system administrators and could mean

that additional system administrators are employed to ensure that the systems do run smoothly.

10.2 Nagios

10.2.1 Description

Nagios is an open source software tool designed for use on Linux systems as a computer system and network monitoring tool. Nagios is a recursive acronym standing for "Nagios Ain't Gonna Insist On Sainthood" [33], although other more descriptive names have been suggested. Nagios provides system administrators with immediate monitoring and notification of problems as well as a basic web interface with which they can view any warnings and details of the problems that have occurred. There are currently two versions of Nagios in use. The first is the open source Nagios platform. The other version is Nagios XI, which is a proprietary solution built on top of the open source version using proven OSS components and a much more detailed web interface including integrated performance graphing, customizable dashboards, a web configuration GUI, and configuration wizards [32].

10.2.2 Comparison of Features

10.2.2.1 Monitoring

Nagios offers monitoring of publicly available services that are provided by Linux, Windows, and Netware servers by default. These publicly available services are generally widely accepted network protocols such as HTTP, FTP, SSH, and SMTP. Monitoring of these services is fairly easy to configure and can be done with the basic Nagios install. Nagios also offers monitoring of advanced features such as memory usage, free disk space, and CPU load. However, these features require a platform specific client to be installed on the target system and are limited to what the platform client can report to the Nagios server.

The greatest distinguishing feature between Nagios and the proposed system is the monitoring of advanced features, such as free disk space and CPU load, on remote systems. Nagios requires the use of third party plug-ins to perform monitoring of this type, whereas

our system only requires an SSH server to which it can connect. In addition, Nagios has no custom triggers that can be used with the advanced feature monitoring and essentially will only report a problem if for example, the CPU usage remains high for an extended period of time.

10.2.2.2 Alerting

Nagios offers an alerting function, similar to that in Big Brother, in the form of a web page. This web page uses colour coded events so that system administrators can quickly attend to critical problems in the systems. Nagios also provides additional functionality including, availability reports that allow system administrators to ensure service level agreements are being met, a historical record of all alerts, notifications, outages, and alert responses, and the availability of third-party add-ons to extend reporting capabilities. In terms of repair ability, Nagios includes a primitive event handler that allows automatic restart of failed applications and services.

In contrast, as stated previously, the proposed system has no actual default alerting features as this was not a design goal of the system. Instead, the system was designed with automatic detection and repair in mind, and only issues alerts if these functions fail. Such alerts include the preliminary diagnostic output to give administrators an initial idea of where the fault might be located. This approach is significantly more advanced than the primitive system included with Nagios in that the proposed system attempts to find the root cause of the fault and repair this, thereby restoring full operation to all systems and services.

10.2.3 Discussion

Nagios is a fairly comprehensive product as it provides system administrators with the ability to monitor systems and services together with alerting features. Nagios also offers a primitive repair function that detects processes that have crashed by monitoring the process list returned from the remote machine and restarts the process if configured to do so, with a process start command. However, this type of primitive repair mechanism is not sufficient for repairing a Grid as the problem may require a more in depth understanding of the Grid system. It is also possible that the repair process of Nagios may not succeed in repairing the root cause of the problem and only remedy a known process crash, for instance, if a process spawned by the Grid crashes. The failure of this spawned process

can cause other services to fail, which are unlikely to be fixed as the spawned process may not be monitored or if it is, it may not be known how it can be restarted.

10.3 Webmin

10.3.1 Description

Webmin is a web-based interface for Unix system administration that allows system administrators to perform tasks on a server remotely using only a web browser [50]. These tasks include the modification of configuration files, creation of user accounts, and remote management of the system from the console. Webmin has a large collection of standard modules that can be downloaded to increase the functionality provided to the system administrator. These standard modules include functionality for bandwidth monitoring and modules for configuration of most common service types used on a server.

10.3.2 Comparison of Features

10.3.2.1 Monitoring

One of the Webmin standard modules is a module called the MON Service Monitor, which allows the system administrator to monitor the availability of services and generate alerts on the occurrence of prescribed events. Since a wide variety of available protocols may be included in the monitoring, the Service Monitor needs to understand many protocols so that it can verify whether a service is working correctly. Webmin can also send alerts to system administrators, which take the form of actions such as sending emails, making submissions to ticketing systems, or triggering resource fail-over in a high-availability cluster.

Although the Webmin MON Service Monitor module can monitor many protocols, a large number of these require the host to support SNMP in order for the monitoring to work. This means that if the SNMP agent stops responding the monitor has no alternative way of determining the status of the system other than a simple ICMP ping. The problem with this becomes most apparent when monitoring unexpected system reboots. If the remote machine reboots and the SNMP agent fails to start then it will not be apparent to system administrators that the machine performed an unexpected reboot as they would simply receive a notification that SNMP was not responding.

10.3.2.2 Alerting

The MON Service Monitor provides a variety of alerting functions to system administrators including email notification, IRC messages, and submissions to the Bugzilla problem-tracking system. These alerts inform system administrators of any problems detected by the module. Webmin also allows easy addition of new alerts to the system, in the form of a simple shell script or any other executable program.

10.3.2.3 Configuration

The main intended use of Webmin is to remotely perform configuration of services running on remote machines. To this effect it has a large selection of modules each of which provide configuration options for one of the many popular services in use today, such as Apache, which is a commonly used web server. Each of these modules provides a web-based configuration tool for the respective service allowing the system administrator to change quickly and easily, any configuration necessary to include new or change existing functionality.

10.3.3 Discussion

Webmin provides a very good product for effective monitoring by system administrators of the systems and services on their network. It does not, however, provide any mechanism to facilitate the duties of the system administrators in terms of fixing the problems that have been detected. Webmin only provides an easy way for them to make configuration changes and a monitoring system to notify them of any problems that have occurred. This means that system administrators still have to connect to the machine directly or via the Webmin interface, to run any commands needed to fix the problems they have been notified of. In other words, Webmin provides additional knowledge and a better understanding of what is happening with the systems and services on the network, but no improved mechanisms for dealing with these issues.

10.4 Summary

Big Brother, Nagios, and Webmin are excellent products and fulfill their roles of monitoring systems and services on a network exceptionally well. They include good reporting abilities

and visual interfaces that allow system administrators to understand quickly what is happening on their networks. This understanding is particularly useful when trying to debug an error occurring on a system or the network. However, they do not provide any significant improvement to simplify a system administrator's job. This is especially important for Grid administrators who may have hundreds, if not thousands, of resources connected to their Grid at any one time.

Due to the high number of machines connected to the Grid, an intelligent system is needed that can repair simple to medium difficulty problems to reduce the number of issues that system administrators routinely have to deal with.

Chapter 11

Conclusion and Future Work

With the shift towards the widespread use of distributed computing systems, grid computing is becoming increasingly important due to its high redundancy, low cost, excellent performance, and high scalability. Grid systems are excellent platforms for customer service deployment due to these factors, while the minimal client side software requirements make them far easier to maintain than older solutions. However, when faults occur on these systems they can be incredibly hard to diagnose and could cause a large amount of downtime or service unavailability owing to the large number of distributed components involved in the system. Possible downtime could be significantly reduced by an automated repair system that monitors the grid and immediately gets to work in attempting to diagnose the cause of any fault. If a diagnosis can be made, an intelligent system can be employed to automatically repair the fault, thus returning the grid to a normal operating state. Such a system would drastically reduce the amount of work performed by the grid system administrators in order to maintain the grid in good working condition, thereby allowing them to concentrate on improving the grid and the services it provides.

In creating an automated grid repair system a number of factors needed to be considered during the design phase so as to produce a system capable of detecting faults occurring on the grid. The most important factor is that the repair system has a fast response time, since each second taken by the repair system to correct a fault is another second that the grid and services it provides are unavailable to consumers. Security is another factor, since the repair system is required to have access to all parts of the grid to effect any necessary repairs. Thus the repair system should not introduce any new security faults and can provide additional security protection via the continuous monitoring. Since the underlying grid system is not exceptionally intrusive and requires minimal client side

software, the repair system should be similarly aligned with minimal software requirements thus correlating well with the grid system requirements. The grid system was designed to maximise processing power by distributing the load over many machines. To maintain this high level of distributed processing power, the repair system does not introduce a high overhead when monitoring the grid. Designing the repair system to be modular also improves the potential for expansion of the system, as well as allowing the repair system to work on multiple platforms and not be dependant on a specific grid technology. Having a completely open-source end-product also ensures that the repair system remains a low cost solution, albeit one that uses well known protocols with no known security faults, thereby reducing the risk of security breaches.

The RAM metric was designed to monitor the memory consumption of the target machine and detect faults by watching for unexpected or large changes in memory usage. In doing so, it also takes into account the target machine's use of the operating system swap file to avoid flagging memory changes where data is simply moved from one memory to the other. Results of the experiments conducted on the RAM metric, show that it is often possible to detect faults that occur on a server just by monitoring for memory changes, and the prototype system effectively detected process crashes and other events such as machine reboots. The results also showed that even with this limited information it was mostly possible to keep the grid in a working condition. However, the RAM metric failed to detect small process crashes when there were a large number of services on a single server that were constantly changing their memory usage. This is an expected side effect of the fact that detection is based purely on memory changes, and can be combated by altering the percentage used for memory detection or changing the detection algorithm to incorporate detections for smaller processes according to a minimum threshold size.

The grid jobs metric was designed to monitor the log file output from the MiG server and observe job submission, execution, completion, and failures for any irregularities or unexpected changes. Based on the experimental results, the grid jobs metric performed well in detecting large unexpected changes in the jobs data through the detection of large spikes and dips. The grid jobs metric was also effective at detecting periods of inactivity on the grid, and checking the system to make sure it was functioning correctly and that the inactivity was not a result of a grid system failure. The repair system was also effective at detecting when a resource started returning a high number of failed jobs to the grid, and although MiG does not support resource disconnection at this time, a rule could easily be added to the firewall to stop the resource from communicating further with the grid until such time as the resource is repaired. The metric also had good success at detecting when a resource stopped requesting jobs from the grid. If the resource was part of the

machine set that was being monitored by the repair system, it was possible for the repair system to connect to the resource and check the resource machine for faults that would cause it to no longer request jobs.

In evaluating the results obtained from the RAM and grid jobs metrics, it was found that they achieved good performance with respect to maximum throughput and response time. Although the response time was slow, it was shown to be relative to the polling interval used by the repair system and thus faster response times could easily be achieved simply by reducing the polling interval until the desired response time is achieved. However, it was also shown that decreasing the polling interval would significantly increase the overhead of the repair system and introduce a greater workload onto the remote system in returning the required information.

The accuracy of the two metrics was also evaluated and it was found that their accuracy in detecting events was fairly good. The RAM metric performed well given that the detection percentage used was optimised for the machine configuration and its average memory usage. Using an optimised percentage the RAM metric was able to detect even small process crashes that occurred on the monitored machine without detecting any fluctuation points. The grid jobs metric was accurate in some respects, however, suffered from a quick falloff if the data set fluctuated violently, owing to the statistical nature of its detection method.

The final evaluation that was undertaken on these metrics was that of testing their effectiveness at repairing the faults that were detected. The RAM metric was very effective in detecting and repairing any faults that occurred and kept the grid system operational even after repeated manual shut downs. The repair process was extremely fast, less than two seconds, once an event had been detected and passed on to the expert system for diagnosis and repair.

Finally the repair system was compared to several other systems that are intended to perform a similar job, but which have not been optimised for grid computing and in fact, have no knowledge at all about grid systems and their composition. All the systems performed fairly well with respect to detecting and reporting errors to the system administrator, however, in most cases, this functionality is provided by primitive tests such as attempting to connect to a port to show that the port is open. In order for these systems to provide more information that these primitive results, operating system specific client-side software is required to be installed on the target machine. Although Nagios includes a repair feature, it is rather primitive in comparison with the expert system

used in the developed protoype and can only repair primitive faults through the use of predetermined scripts or commands.

From the above discussion, we conclude that our attempt at creating an automated repair system, with innate knowledge on how grid systems operate and which can monitor and repair grid systems using a least intrusive approach, has been successful. The prototype repair system is able to detect and repair multiple types of faults that occur with acceptable performance, accuracy, and effectiveness. Based on the prototype, a production system needs be implemented and put into use to ease the maintenance of current grid systems and reduce downtime and lack of productivity as a result of faults occurring.

The work carried out in this thesis represents an initial investigation into the effectiveness of creating an automated grid repair system, and as such there are opportunities to expand on the findings of this research. Firstly, additional metrics need to be created to obtain better coverage of all features of the monitored machine. Secondly, more complex detection algorithms could be created and used with the current metrics, or new metrics, to detect events with greater accuracy. Additionally, an investigation into the most effective algorithm approaches can be undertaken so as to provide recommendations for each metric.

Bibliography

[1] ACM. The ACM Computing Classification System 1998 Version. Online, Dec. 2011. Available from: http://www.acm.org/about/class/ccs98-html.

[2] Ars Technica. Another fraudulent certificate raises the same old questions about certificate authorities. Online, Nov. 2011. Available from: http://arstechnica.com/security/news/2011/08/earlier-this-year-an-iranian.ars.

[3] Ars Technica. Independent Iranian hacker claims responsibility for Comodo hack. Online, Nov. 2011. Available from: http://arstechnica.com/security/2011/03/independent-iranian-hacker-claims-responsibility-for-comodo-hack.ars.

[4] Bayer, M. SQLAlchemy Documentation. Online, Oct. 2011. Available from: http://www.sqlalchemy.org/docs/contents.html.

[5] Ben-Gal, I. E. *Encyclopedia of Statistics in Quality and Reliability*. John Wiley & Sons, 2007.

[6] Bender, E. A. *Mathematical Methods in Artificial Intelligence*. IEEE Computer Society Press, 1996.

[7] Big Brother. Big Brother System and Network Monitor - Features. Online, Oct. 2011. Available from: http://www.bb4.org/features.html.

[8] Casanova, H. Distributed computing research issues in grid computing. *SIGACT News 33*, 3 (2002), 50–70.

[9] Clabby Analytics. The grid report. Tech. rep., Clabby Analytics, 2004.

[10] Dierks, and Rescorla. RFC2546 : The Transport Layer Securty (TLS) Protocol Version 1.2. Online, Oct. 2011. Available from: http://tools.ietf.org/html/rfc5246.

[11] Dumpleton, G. [mod_python] mod_python and mod_wsgi. Online, July 2007. Retrieved on 06 May 2010. Available from: `http://www.modpython.org/pipermail/mod_python/2007-July/024080.html`.

[12] Foster, I. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG* (2002).

[13] Foster, I. What is the Grid? A Three Point Checklist. *Grid Today 1*, 6 (July 2002), 22–25.

[14] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing* (2005), Springer-Verlag LNCS 3779, pp. 2–13.

[15] Foster, I., and Kesselman, C., Eds. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[16] Foster, I., Kesselman, C., and Tuecke., S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal Of Supercomputer Applications 15*, 3 (2001), 2001.

[17] Freier, Karlton, and Kocher. RFC6101 : The Secure Sockets Layer (SSL) Protocol Version 3.0. Online, Aug. 2011. Available from: `http://tools.ietf.org/html/rfc6101`.

[18] Globus.org. Globus Toolkit 5.2.0 Release Manuals. Online, Dec. 2011. Available from: `http://www.globus.org/toolkit/docs/latest-stable/`.

[19] instagrid.org. Instant-Grid — Ein Grid-Demonstrations-Toolkit. Online, May 2010. Available from: `http://www.instant-grid.org/`.

[20] Jackson, P. *Introduction To Expert Systems*. Addison Wesley, 1998.

[21] Langtangen, H. P. *Python Scripting for Computational Science*, 3rd ed. Springer, 2008.

[22] Li, M., and Baker, M. *The Grid: Core Technologies*. John Wiley & Sons, 2005.

[23] Lloyd, J. W. *Foundations of logic programming*. Springer-Verlag, 1984.

[24] MacGuire, S. Big brother - a web-based system & network monitor. In *SysAdmin, Audit, Networking, and Security Conference* (1998).

[25] MacGuire, S., and Croteau, R.-A. Big brother is still watching - a web-based system & network monitor. In *SysAdmin, Audit, Networking, and Security Conference* (1999).

[26] Martelli, A. *Python in a Nutshell, (2nd Edition).* O'Reilly, 2006.

[27] Microsoft TechNet. SSL/TLS in Detail. Tech. rep., Microsoft, 2003.

[28] Mitchell, T. M. *Machine Learning.* McGraw-Hill Book Company, 1997.

[29] modpython. mod_python Apache/Python Integration. Online, May 2010. Available from: `http://modpython.org/`.

[30] modpython. mod_python Project Tracker. Online, May 2010. Available from: `https://issues.apache.org/jira/browse/MODPYTHON`.

[31] modwsgi. modwsgi: Python WSGI adapter module for Apache. Online, May 2010. Available from: `http://code.google.com/p/modwsgi/`.

[32] Nagios. Nagios - The Industry Standard in IT Infrastructure Monitoring. Online, Oct. 2011. Available from: `http://www.nagios.org/`.

[33] Nagios Support. Nagios - What does it mean? Online, Oct. 2011. Available from: `http://support.nagios.com/knowledgebase/faqs/index.php?option=com_content&view=article&id=52&catid=35&faq_id=2&expand=false&showdesc=true`.

[34] NorduGrid. NorduGrid. Online, May 2010. Available from: `http://www.nordugrid.org/`.

[35] Oracle Corporation. MySQL 5.5 Reference Manual. Online, Nov. 2011. Available from: `http://dev.mysql.com/doc/refman/5.5/en/`.

[36] php.net. PHP: FAQ. Online, Apr. 2010. Available from: `http://www.php.net/manual/en/faq.php`.

[37] Pyke. PyKE - Python Knowledge Engine. Online, Nov. 2011. Available from: `http://pyke.sourceforge.net/`.

[38] Python.org. About Python. Online, Apr. 2010. Available from: `http://www.python.org/`.

[39] Python.org. Python FAQ. Online, Apr. 2010. Available from: `http://www.python.org/doc/faq/`.

[40] Python.org. Python v3.1.2 documentation. Online, May 2010. Available from: `http://docs.python.org/py3k/`.

[41] Sanner, M. F. Python: A Programming Language For Software Integration And Development. *J. Mol. Graphics Mod 17* (1999), 57–61.

[42] Schnier, B. 10 Risks of PKI. *Computer Security Journal XVI* (2000).

[43] SETI@home. SETI@home. Online, May 2010. Available from: `http://setiathome.berkeley.edu/index.php`.

[44] Stewart, J. *Calculus: Early Transcendentals 5th Edition*. Brooks Cole, 2002.

[45] Trubetskoy, G. Introducing mod_python. Online, Feb. 2003. Retrieved on 06 May 2010. Available from: `http://onlamp.com/pub/a/python/2003/10/02/mod_python.html`.

[46] Vinter, B. The Architecture of the Minimum intrusion Grid: MiG. In *in proc of Communicating Process Architectures* (2005), IOS Press, pp. 189–201.

[47] Vinter, B. Minimum Intrusion Grid. Online, Mar. 2010. Available from: `http://sites.google.com/site/minimumintrusiongrid/`.

[48] Vinter, B., Andersen, R., Rehr, M., Bardino, J., and Karlsen, H. Towards A Robust And Reliable Grid Middleware. In *Grid Technology and Applications: Recent Developments* (2009), Nova Science Publishers, Inc.

[49] Walker, D. The grid, virtual organizations, and problem-solving environments. *Cluster Computing, IEEE International Conference on Cluster Computing* (2001), 445–447.

[50] Webmin. Webmin. Online, Jan. 2011. Available from: `http://www.webmin.com/`.

[51] Zissis, D., and Lekkas, D. Addressing cloud computing security issues. *Future Generation Computer Systems In Press,*, 3 (2010), 583–592.