THE UNIVERSITY OF CALGARY

Monitoring Jobs in Grid Computing Environments

by

Idowu Oluwafemi Adewale

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2007

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "**Monitoring Jobs in Grid Computing Environments**" submitted by Idowu Oluwafemi Adewale in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Supervisor, Dr. Rob Simmonds
Department of Computer Science

[illegible]

Dr. Denilson Barbosa
Department of Computer Science

[illegible]

Dr. Abraham Fapojuwo
Department of Electrical and
Computer Engineering

19/12/2007

Date

Abstract

In order to improve the performance of jobs in a grid computing environment, it is important to have a job monitoring system that could show users how their jobs behaved during execution on a computer. Most of the existing job monitoring tools report on the status of scheduled jobs, but not on the behaviour. Recent tools that report on behavioural aspects of jobs have done so in cumbersome manners.

This thesis develops a technique, known as the *Wrapper method*, for monitoring jobs in grid computing environments. We developed a monitoring tool to implement the Wrapper technique; the tool monitors jobs on a computer and collects monitoring data about the jobs. We also implement a graphing tool for converting monitoring data into meaningful monitoring information.

We report on series of experiments to validate the job monitoring capabilities of the *Wrapper method*. The results show that memory, local disk I/O, network, and shared file system can be bottlenecks to the performance of a job. We suggest a number of ways to improve the performance of jobs affected by these bottlenecks.

Acknowledgements

First and foremost, I thank God for giving me the opportunity, patience, grace, and strength to complete this research work.

I am greatly indebted to my supervisor, Dr. Rob Simmonds, for giving me the opportunity to do my Masters thesis under his supervision. Your guidance, expertise, and feedback have helped in bringing this thesis to completion.

Thank you to my examiners - Dr. Denilson Barbosa (Internal) and Dr. Abraham Fapojuwo (External) for reading through my thesis and providing valuable feedback.

The Grid Research Centre group at the University of Calgary has been a large source of help during the course of my program. In particular, I would like to thank Cameron Kiddle, Roger Curry, Abhishek Gaurav, and Mark Fox for their help during the course of this research.

A special thanks to my lovely wife, Opeyemi, for all her support. The story of my life would not be complete without you. Your love, prayers, and encouragement makes my living worthwhile. My special gratitude is due to my mum and brother, just as I equally appreciate the support of my sisters and their families. I feel a deep sense of gratitude to my late father who laid the foundation for my education and supported me with all the materials to enable me succeed academically.

I am grateful to my wonderful friend - Omolade Saliu. You have been a source of advice, motivation, and encouragement during my Masters programme; you have touched my life in various ways. I am tempted to mention all the friends that have encouraged me along the way; the list would be endless and I am afraid I might leave someone out. I am using this opportunity to simply say "Thank you" to you all. However, I wish to thank Bashiru Ikharia, Ohi Imoukhuede, Joseph Amuah, Kenny Oladosu, Gbenga Adeyanju, Tosin Alao, Femi Ogunsuyi, Sunday Fagbemiro, Emmanuel Ibidokun, Yinka Odebade, Olumide Adeoye, Nina Ejezie, and Laide Olorunleke for their encouragement.

My gratitude goes to Paul and Nancy Starling, Abiodun and Abisoye Fatokun, the Ajalas, the Omotayos, the Adeyanjus, and the lovely people at UChurch for all their encouragement during my Masters programme.

I would always be grateful to Remi Okeyinka for being my motivator during my undergraduate programme at Ladoke Akintola University of Technology, Ogbomoso, Nigeria. Your motivation have brought me this far.

The generous support from the Department of Computer Science and the Grid Research Centre at the University of Calgary is greatly appreciated.

Table of Contents

List of Figures

Chapter 1

Introduction

Grid computing allows users to access and share powerful computers, databases, computing storage facilities, high-speed fibre optic links, network resources, and experimental facilities across different geographic areas or administrative domains or both. The sharing of resources in grid computing environments happens transparently with the user not necessarily knowing where those facilities are located. The sharing is concerned with direct access to computers, software, data, and other resources, as required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering [58].

Grid computing is used to solve research and business problems that require a large amount of computational resources. Some of these problems are so complex and data-intensive that a large amount of computational power (which may include powerful computers, large amounts of memory, and high data transfer speed) is needed to solve such problems. Grid computing is helping organizations to leverage existing hardware investment and resources, reduce operational cost, accelerate product development, and increase productivity [5]. Grid computing has commercial applications in different industries including aircraft engine design, crash test simulation [38, 48], telecommunication network planning and management [72], financial modelling using Monte Carlo simulations [33, 46, 91], digital rendering and animations in the entertainment industry, computational fluid dynamics in the manufacturing industry, and 3D seismic processing in the oil and gas industry [6]. The importance of grid computing in the research community is immense. Grid computing is useful in many scientific problems which require a massive amount of computation [48, 77, 82]. In addition, it is helping researchers and scientists to interact and share data, instruments, and visualization tools irrespective of their geographical location.

1.1 Grid Monitoring

Monitoring in grid environments is the act of collecting information concerning the characteristics and status of resources of interest [93]. In [56], grid monitoring is defined as, "the measurement and publication of the state of a grid component at a particular point in time." The discovery, characterization, and monitoring of resources, services, and computations can be challenging due to the considerable diversity, large numbers, dynamic behaviour, and geographical distribution of the entities in which a user might be interested [56]. Monitoring is required for a number of purposes, including status checking, troubleshooting, performance analysis and tuning, debugging, auditing, and intrusion detection [13, 32].

A job is defined as, a unit of work defined by a user; it may include a set of computer programs, files, and control statements to the computer operating system [14]. A job requests required resources and gets the resources when they are available. A batch job is often defined using a shell script. The batch script once created for a job can be reused or modified to start the job as many times as possible.

A job is classified as sequential or parallel depending on whether the job is to be processed on a sequential computer or on a parallel computer with several nodes. Parallel jobs consist of multiple processes that run concurrently on a set of processors [55]. In all cases there is a need for the programs in a parallel job to interact with each other by exchanging data through Inter Process Communication (IPC). In addition, the processors executing a parallel job may be on the same computer or on different computers. Sequential jobs are scheduled to execute on a sequential computer.

1.2 Problem Statement

When a job is submitted for execution in a grid computing environment, monitoring becomes essential so that a user can see that the job was completed in an expected way within the expected time frame. It is also necessary that the user should be able to detect any problem that occurs while the job is running. In a grid environment, a user loses direct control over the job after it has been submitted, that is the user does not know what is happening to the job on the computer on which the job is being executed [7].

One of the challenges in grid computing is how to measure the performance of grid infrastructure and grid applications. There is no widely accepted and deployed technique that can solve all aspects of the problem [40]. In order to answer the question of grid performance, numerous approaches to performance monitoring and evaluation have resulted in several different tools. The popular grid monitoring tools used in the industry and academia are Monalisa [54], Ganglia [31], and Netlogger [51]. These tools give accurate information on basic system configuration, memory statistics, and CPU usage statistics, but they do not give information about individual jobs.

Monalisa reports the number of running and queued jobs on a computer, but does not give any information on the node(s) on which the jobs are being executed. Also, the JINI server in California Institute of Technology must be up and running before a user can monitor any computer running Monalisa. Ganglia reports the basic information about the state of a computer on a web page, and uses RRDtool (a round robin database) [31] for data storage and visualization. The round-robin database does not keep a long history; hence, a user can only view reports of the last 24 hours, 1 day, 1 week, 1 month and 1 year. It does not report any information about the jobs running on a computer. In addition to reporting memory and CPU usage statistics, Netlogger could be used to instrument other applications like Grid File Transfer Protocol (GridFTP) [75] to monitor file transfers, but it does not report information about individual jobs on a computer.

1.3 Motivation

Most of the existing job monitoring tools report on the status of scheduled jobs, but not on the behavioural aspects. Recent tools that report on behavioural aspects of jobs have done so in cumbersome manners. When users submit jobs in a computing environment, in most cases they would like to know how their jobs are performing on a computer. For example, is their job getting enough processing time, is their job getting enough memory, is their job doing a large amount of disk Input/Output operations, is their job using a large amount of shared resources like network bandwidth and shared file system, and is their job experiencing competition from other activities on the same computer. There are users that are interested in how jobs perform on computers, so they could know where to submit their jobs in the future.

1.4 Thesis Objectives

The following are the main objectives of this research work:

- To perform a survey of the existing grid monitoring tools and approaches, and analyze them from job monitoring perspective. This will help to clearly identify their short comings, thereby opening directions for improvement.
- To design a technique for monitoring jobs in grid computing environment; the technique would be simple and easy to implement.
- To develop a monitoring tool to implement the job monitoring technique designed in this thesis.
- To assist users in understanding the performance of their jobs in a grid computing environment, by providing meaningful monitoring information. The monitoring information would show the bottleneck(s) to the performance of a job and suggest a number of ways to improve the performance of jobs in the future.

1.5 Thesis Overview

The rest of the thesis is organized as follows. Chapter 2 highlights relevant background information on grid computing including a brief history of grid computing, types of grid, grid architecture, and grid components. Chapter 3 discusses some related work and existing grid monitoring tools. The related work discussed includes the Grid Monitoring Architecture (GMA) [12] and some grid monitoring tools including Ganglia, Monalisa, and Netlogger. The chapter discusses the related work from job monitoring perspective in the context of this thesis work.

Chapter 4 focuses on job monitoring in grid computing environments. This chapter describes a technique for monitoring jobs called *Wrapper method.* The design and implementation of the *Wrapper method* are described in detail. The chapter also discusses job monitoring information, sources of job monitoring information, and how the monitoring information is transferred and presented to users. The design issues with the *Wrapper method* and how it is implemented in a grid computing environment are highlighted. In Chapter 5, the results of eight experiments are presented and discussed. The purpose of the experiments is to identify the bottlenecks to the performance of jobs on a computer under different scenarios. The experimental design, the workload, and the environmental setup for the experiments are also described in this chapter. Chapter 6 concludes the thesis by summarizing the contributions of this research work, and suggesting future research directions.

Chapter 2

Background

This chapter presents a brief overview of grid computing and grid computing technologies relevant to this thesis. It starts with a brief history of grid computing in Section 2.1 and discusses the different types of grid computing environments in Section 2.2. The type of environment under which this thesis is performed is highlighted in this section. Section 2.3 provides an overview of grid architecture while the Globus Toolkit [35, 43] is presented in Section 2.4. Section 2.5 describes the key grid computing components. Finally, the chapter is summarized in Section 2.6.

2.1 Brief History of Grid Computing

In 1969, the US Defense Department's Advanced Research Projects Agency (ARPA) created the Advanced Research Projects Agency Networks (ARPANET). ARPANET was designed to be a system of data communications for scientific and military operations that could withstand nuclear attack. ARPANET's founders designed it so that authority was distributed over a large number of geographically dispersed computers [80]. ARPANET served as a testbed for new networking technologies, linking many universities and research centres. The first two nodes that formed the ARPANET were University of California and the Stanford Research Institute, followed shortly thereafter by the University of Utah. ARPANET was used by a few computer scientists and the Department of Defense (DoD) community [96].

Various production and research networks evolved from ARPANET including

NSFNET (National Science Foundation Network) which was created in 1986 with a 56 kilobit/sec backbone bandwidth that tied together five NSF (National Science Foundation) supercomputer centres. In 1995, the NSF transferred NSFNET to the commercial sector, which later evolved into today's Internet [96]. Today the Internet is a global network connecting millions of computers which enables people to exchange data, news, and opinions. The NSF created NSFNET, in order to give scientific researchers easy access to its new supercomputer centres.

In the early-to-mid 1990s, a number of research projects in the academic and research community focused on distributed computing. Distributed computing is the process of aggregating the power of computing entities to collaboratively run a single computational task in a coherent way, so that they appear as a single, centralized system [106]. Some research focus is on methods of dividing computational jobs into smaller pieces for multiple machines. One key area of research focused on developing tools that would allow distributed high performance computing systems to act like one large computer.

At the 1995 supercomputing conference sponsored by the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery (IEEE/ACM), 11 high speed networks were used to connect 17 sites with high-end computing resources for a demonstration to create one super "metacomputer." This demonstration was called I-Way. Sixty different applications, spanning various faculties of science and engineering were developed and run over the demonstration network. Many of the early grid computing concepts were explored in the demonstration as the team created various software programs to make all computing resources work together [5].

The demonstration of the I-Way was successful and people were convinced that grid computing had great potential. In October 1996, the U.S. Defense Advanced Research Projects Agency (DARPA) funded a project to create foundation tools for distributed computing. The research project was led by Ian Foster of Argonne National Laboratory (ANL) and Carl Kesselman of University of Southern California. At the 1997 supercomputer conference, Foster and Kesselman demonstrated a grid with some 80 sites worldwide running Globus Toolkit [43] middleware. At that point, Foster and Kesselman had started calling it "grid computing," playing on the analogy to the electrical grid [105].

2.2 Types of Grid

Grid computing vendors have adopted various nomenclatures to explain and define the different types of grids. Some define grids based on the structure of the organization (virtual or otherwise) that is served by a grid, while others define it by the principle resources used in a grid [5].

Considering the structure or size of a computing grid, the three main types of grids are Departmental, Enterprise, and Global grids.

Departmental Grids are deployed to solve problems for a particular group of people within an organization. The resources are not shared by other groups within the organization.

Enterprise Grids have resources spread across an organization and provides service to all the groups and users within the organization. The resources in an Enterprise grid span multiple departments or projects. Enterprise grids enable multiple projects or departments to share resources within an enterprise or campus, and don't necessarily have to address the security and global policy management issues associated with global grids [23].

Global Grids are organizations over the public Internet. They are established to facilitate business or collaboration between the organizations. The services could be purchased in part or in whole from service providers [5]. Global grids are collections of enterprise and cluster grids as well as other geographically distributed resources, all of which have agreed upon global usage policies and protocols to enable resource sharing [23].

This thesis is implemented on *Grid Research Centre cluster* (*grc cluster*) in Grid Research Centre at the University of Calgary. In terms of size, GRC Cluster can be classified as a Departmental grid. It serves as a testbed for developing and testing grid computing and High Performance Computing (HPC) applications.

Grid computing can be used in a variety of ways to address various kinds of application requirements. Often, grids are categorized by the type of solutions they best address [59]. There are no hard boundaries between these grid types and often a grid may be a combination of two or more other types of grid [68]. Using this criterion, the three primary types of grids are described as follows.

2.2.1 Compute Grid

A compute grid environment consists of one or more hardware- and software-enabled environments that provide dependable, consistent, pervasive, and inexpensive access to high end computational capabilities. A compute grid sets aside resources specifically for computing power. It denotes a hardware and software infrastructure that enables coordinated resource sharing within dynamic organizations consisting of individuals, institutions, and resources [83].

In [5], compute grids are classified further by the type of computational hardware used in the grid computing environment. The different types of computational grids include Desktop grids, Server grids, and High-Performance grids. The three primary types of compute grids are described in the next sections.

2.2.1.1 Desktop Grids

The ad hoc collections of work-based and home-based PCs from around the world are an example of PC-based distributed computing and serve as the forerunners of today's Desktop grids [67]. The aggregation of PC processing power became known in the last few years, through one of the many "cause computing" projects.

A Desktop grid is also known as a Scavenging grid in [66]. The machines in Desktop grids are scavenged for available CPU cycles and other resources. Owners of the desktop machines are usually given control over when their resources are available to participate in a grid.

The examples of a Desktop grid include SETI@home [94] - a scientific experiment that uses Internet-connected computers in Search for Extraterrestrial Intelligence (SETI), Greater Internet Mersenne Prime Search (GIMPS) [45] - searching for extremely large prime numbers, and ClimatePrediction.net [16] - for predicting climate on a global scale in the 21st century.

2.2.1.2 Server Grids

In some organizations, special servers are bought solely for the purpose of creating an internal "utility grid" with resources made available to various departments. No desktops are included in server grids. These servers usually run some flavor of the

UNIX/Linux operating system [5].

2.2.1.3 Cluster Grids

Cluster grid is a term used by Sun Microsystems and consists of one or more systems working together to provide a single point of access to users [73]. In [86], Gregory Pfister defines a cluster as "a type of parallel or distributed system that consists of a collection of interconnected whole computers, and is used as a single, unified computing resource." A cluster grid is a superset of other technical compute resources such as Linux clusters, throughput clusters, midrange compute servers, and high-end shared-memory systems. Therefore, the cluster grid can operate within a heterogeneous environment with mixed server types, mixed operating environments, and mixed workloads [23].

2.2.2 Data Grids

A data grid is responsible for housing and providing access to data across multiple organizations. Users are not concerned with where data is located as long as they have access to the data. For example, two universities doing life science research may each have their own unique data. A data grid would allow them to share their data, manage the data, and manage security issues such as who has access to what data [68].

Data grids provide transparent, secure, and high-performance access to federated data sets across administrative domains and organizations. Users (both people and applications) may be unaware that they are using a data grid [6]. The remote data may be flat-file data, relational data, or streaming data [50].

2.2.3 Utility Grids

In [5], utility grids are defined as commercial compute resources that are maintained and managed by a service provider. Customers that have the need to augment their existing, internal computational resources may purchase "cycles" from a utility grid. Customers may choose to use utility grids for business continuity and disaster recovery purposes in addition to overflow applications.

This thesis work is carried out on a cluster grid known as *grc cluster*. It provides resources for developing and running scientific computing applications. It has 11 nodes connected together to form a powerful computer.

2.3 Grid Architecture

A computing grid architecture identifies fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another. Ian Foster et al. proposed grid architecture in [58]; they defined grid architecture from the perspective that sharing relationships need to be established among potential participants, for an organization to function effectively. The grid architecture is a protocol architecture with protocols defining the basic mechanisms by which Virtual Organization (VO) users and resources negotiate, establish, manage, and exploit sharing relationships. The standard-based open architecture facilitates extensibility, interoperability, portability, and code sharing. The standard protocols also make it easy to define standard services that provide enhanced capabilities.

Interoperability is key in order to ensure that sharing relationships can be initiated among arbitrary parties, accommodating new participants dynamically, across different platforms, languages, and programming environments. Mechanisms are implemented so as to have interoperability across organizational boundaries, operational policies, and resource types. The grid architecture components are organized into layers as shown in Figure 2.1.

2.3.1 Grid Architecture Description

The description of grid architecture does not provide a complete list of all required protocols (and services, APIs, and SDKs) but rather to identify requirements for general classes of components [58]. The principles of the "hourglass model" [19] are used in specifying the various layers of the grid architecture. In the proposed architecture, the neck of the hourglass consists of *Resource and Connectivity* protocols, which facilitate the sharing of individual resources. The Resource and Connectivity protocols are designed so that they can be implemented on top of a diverse range of resource types defined at the *Fabric layer.* They can also be used to construct a wide range of global services and application-specific behaviours at the *Collective layer.* The Collective layer involves the coordinated ("collective") use of multiple resources.

The grid architecture shown in Figure 2.1 has been closely aligned with the Internet protocol architecture as defined by the Open Systems Interconnect (OSI) Internet stack [5]. Protocols, services, and APIs occur at each level of the grid architecture model. The components within each layer share common characteristics but can build on capabilities and behaviours provided by any lower layer component. The grid architecture components are discussed in subsequent sections.

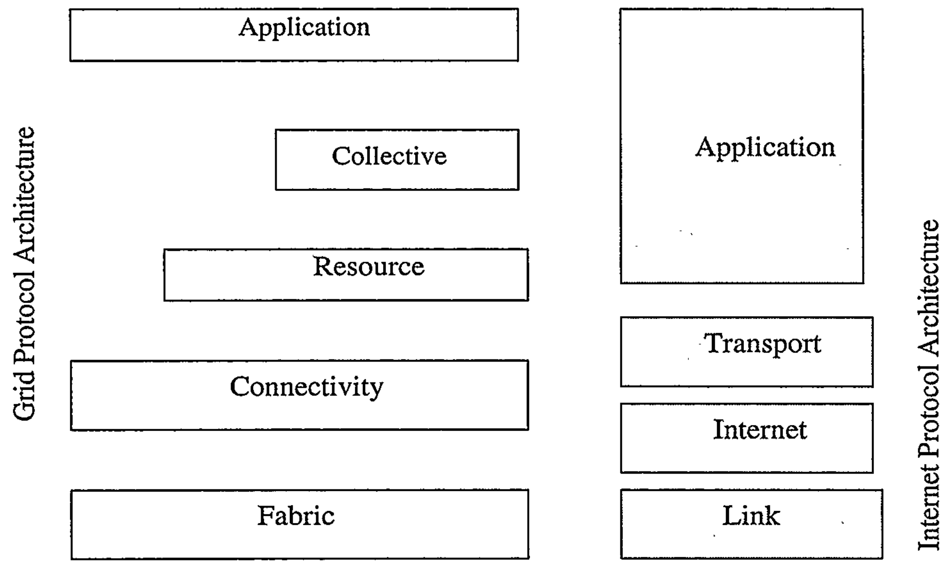


Figure 2.1: The layered Grid Architecture and its relationship to the Internet Protocol (Source [58])

2.3.1.1 Fabric Layer

The grid architecture Fabric layer includes the protocols and interfaces that provide access to the resources that are being shared. Examples of shared resources are computational resources, storage systems, code repositories, catalogs, network resources, and sensors.

2.3.1.2 Connectivity Layer

The connectivity layer defines core communication and authentication protocols required for grid-specific network transactions. The communication protocols enable the exchange of data between Fabric layer resources.

2.3.1.3 Resource Layer

The Resource layer is built on the communication and authentication protocols of the Connectivity layer. This layer defines protocols for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations, on individual local resources. The protocols defined at this layer include

Grid Resource Allocation Management (GRAM) - used for remote allocation, reservation, monitoring, and control of computational resources.

Grid File Transfer Protocol (GridFTP) - for high performance data access and transport.

Grid Resource Information Service (GRIS) - grants access to structure and state information.

2.3.1.4 Collective Layer

In contrast to the Resource layer that focus on interactions with a single resource, the Collective layer contains protocols and services that are global in nature, and capture interactions across collections of resources.

2.3.1.5 Applications Layer

This layer defines protocols and services that are targeted toward a specific application or a class of applications. This layer comprises the user applications that operate within a VO environment.

The *Wrapper method* described in Chapter 4 of this thesis belongs to the Resource layer of the grid architecture and falls in the Application layer of the Internet Protocol architecture. It is important to know the position of the *Wrapper method* in the grid architecture, for the purpose of interoperability.

2.4 Globus Toolkit

Globus [42] is a community of users and developers who collaborate on the use and development of open source software, and associated documentation, for distributed computing and resource federation. The Globus Toolkit [35] is a middleware available under an open-source license from the Globus Alliance consortium [44].

The Globus Toolkit is a fundamental enabling technology for the "Grid," letting people share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit includes software services and libraries for security, information infrastructure, file management, resource management, data management, communication, fault detection, and portability [41]. The toolkit is a central part of science and engineering grid computing projects, and an underlying layer on which leading IT companies are building significant commercial grid products.

The Globus Toolkit was conceived to remove obstacles that prevent seamless collaboration. Its core services, interfaces, and protocols allow users to access remote resources as if they were located within their own machine room while simultaneously preserving local control over who can use resources and when they can access the resources [37].

The Globus Toolkit has evolved rapidly from version 1.0 in 1998 to the 2.0 release in 2002 and now the latest version 4.0. The latest version of Globus Toolkit known as GT4 is based on new open-standard grid services. The Globus Toolkit is the de facto standard for open source grid computing infrastructure. Globus Toolkit provides a variety of components and capabilities, including the following:

- A set of service implementations focused on infrastructure management.

- Tools for building new Web services, in Java, C, and Python.
- A powerful standards-based security infrastructure.
- Client APIs (in different languages) and command line programs for accessing various services and capabilities.
- Detailed documentation on the various Globus Toolkit components including their interfaces, and how they can be used to build applications.

The description of Globus Toolkit in this section focuses on the Web services-based GT4. GT4 makes extensive use of *Web Services mechanisms* to define its interfaces and structure its components. Web services provide flexible, extensible, and widely adopted XML-based mechanisms for describing, discovering, and invoking network services; in addition, its document-oriented protocols are well suited to the loosely coupled interactions preferable for robust distributed systems [34].

Since the Globus toolkit is the de facto standard for grid computing, it is important to have a basic overview of Globus toolkit. How jobs are monitored in a grid computing environment with Globus middleware using Globus Toolkit is described in Section 4.8 of Chapter 4.

2.5 Grid components

This section highlights the key components that make up a grid environment. The most common description of a computing grid includes an analogy to a power grid. When an electrical power consumer plugs an electrical appliance into a receptacle, he/she expects power to be available at the correct voltage. The user does not know the actual source of the power. The local utility company provides the interface into a complex network of generators and power sources and provides consumers with the correct power based on their energy demands. Rather than each house or neighbourhood having to obtain and maintain its own generator of electricity, the power grid infrastructure provides a virtual generator. The generator is highly reliable and adapts to the power needs of the consumers based on their demand [66].

The vision of grid computing is similar in the sense that once the proper kind of infrastructure is in place, a user will have access to the computing infrastructure that is reliable and adaptable to the user's needs. A computing grid consist of many diverse computing resources, but these individual resources will not be visible to the user, just as the consumer of electric power is unaware of how his electricity is being generated. Depending on the design of a grid and its expected use, some of the components may or may not be required, and in some cases they may be combined to form a hybrid component [65]. Figure 2.2 shows some of the key components that make up a typical grid and the components are discussed in the subsequent sections.

2.5.1 Security

In any grid computing environment there must be mechanisms to provide security, including authentication, authorization, and data encryption. The computing resources in grid computing environments are hosted in different security domains and heterogeneous platforms. Hence, the grid middleware must address local security integration, secure identity mapping, secure access/authentication, secure federation, and trust management [68]. The Grid Security Infrastructure (GSI) component of the Globus Toolkit provides robust security mechanisms. It provides a single sign-on mechanism so that once a user is authenticated, a proxy certificate is created and used to reduce the number of times a user must enter his/her pass phrase when

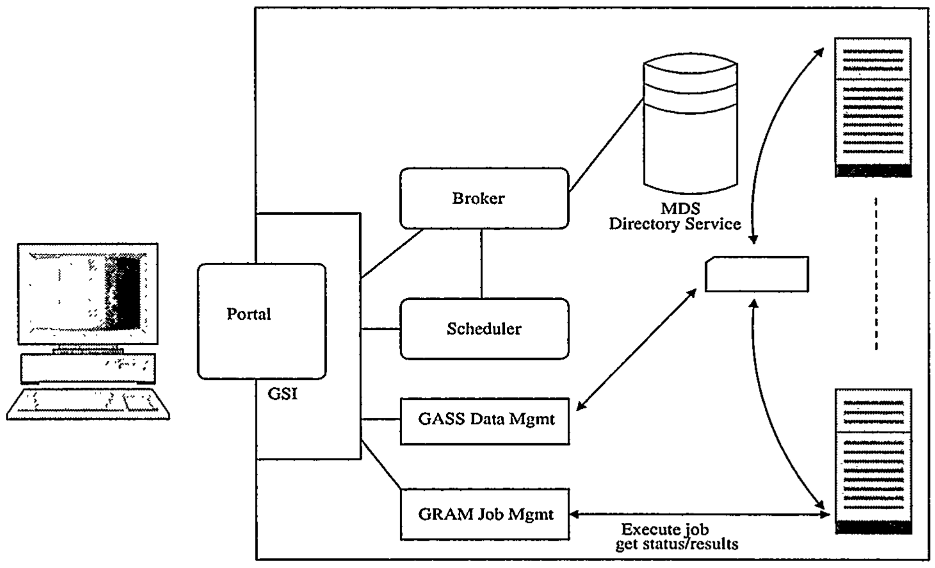


Figure 2.2: Key Grid Components

performing mutual authentication within a grid computing environment [66].

2.5.2 Resource Discovery

After a user is authenticated, there is need to identify the available and appropriate resources to utilize within a grid for the user's job. This task is handled by Monitoring and Discovery Service (MDS). The Globus Resource Allocation and Management (GRAM) processes the requests for resources for remote application execution, allocates the required resources, and manages the active jobs. It also returns updated information regarding the capabilities and availability of the computing resources to the MDS [101].

A standardized GRAM interface gives access to a variety of local resource management tools that a site might have in place, such as Load Sharing Facility (LSF), Network Queuing Environment (NQE), LoadLeveler, Portable Batch System (PBS), and Condor. MDS provides tools to enable the discovery and querying of system components [37]. The service provides information about the available resources within a grid and their status.

2.5.3 Scheduler

When the available and appropriate resources have been identified, the next step is to schedule the individual jobs to run on the resources. This step is handled by the schedulers. Schedulers are types of applications responsible for the management of jobs, such as allocating resources needed for any specific job, partitioning of jobs to schedule parallel execution of tasks, data management, event correlation, and service-level management capabilities [68].

There could be different levels of schedulers within a grid environment. For example a cluster within a grid environment could be represented as a single resource. The cluster may have its own scheduler to help manage the nodes it contains. A higher level scheduler (known as meta-scheduler) might be used within a grid environment to schedule work to be done on a particular cluster, while the cluster's scheduler would handle the actual scheduling of work on the cluster's individual nodes.

2.5.4 Data Management

Data in a grid computing environment may be an input into a resource and/or the results from the resource after a specific task is completed. The data needed by a job may be close to or far from the computation site where it is to be used. Data movement in any grid computing environment requires secure and reliable data transfers, between different computing sites.

The Globus Toolkit contains data management components that provide such services. Some of the data management components are Grid File Transfer Protocol (GridFTP), Reliable File Transfer (RFT), and Replica Location Service (RLS). GridFTP is built on top of the standard FTP protocol, but optimized for high-bandwidth wide-area networks. It utilizes the GSI for user authentication and authorization. Hence, once a user has an authenticated proxy certificate, he can use the GridFTP facility to move files without having to go through a login process to every node involved. This facility provides third-party file transfer so that one node can initiate a file transfer between two other computing sites [66]. The standard FTP protocol has been extended while preserving interoperability with existing servers to develop GridFTP. The extensions provide for parallel data channels, partial files, automatic and manual TCP buffer size settings, progress monitoring, and extended restart functionality [5].

The RFT service provides the reliable management of multiple GridFTP transfers. The RLS is a scalable system for maintaining and providing access to information about the location of replicated files and datasets.

2.5.5 Job and Resource Management

The large number and the heterogeneous nature of grid computing resources makes resource management a great challenge in grid computing environments. Resource management scenarios often include resource discovery, resource inventories, fault isolation, resource provisioning, resource monitoring, a variety of autonomic capabilities, and service-level management activities [68].

The key grid components used in the implementation of the *Wrapper method* are resource manager and scheduler. Portable Batch System (PBS) is used for job and resource management; it is chosen over other batch systems because it is the most popular of all batch systems. And MAUI [18] cluster scheduler is used because it is an open source scheduler for clusters and supercomputers. The experimental testbed is a cluster of computing nodes, so PBS and MAUI are suitable for such environment.

2.6 Summary

This chapter presented a brief overview of grid computing and grid computing technologies relevant to this thesis. Grid computing environments were grouped into Departmental, Enterprise, and Global grids considering their structure or size. The types of grids were also categorized into Compute, Data, and Utility grids based on the type of solutions they provide. The Compute grid category is further divided into Desktop, Server, and Cluster grids based on the type of computational hardware deployed in the grid environment.

The grid architecture that identifies the key fundamental system components of a computing grid was described. It specifies the purpose and function of the components and describes how these components interact with one another. A brief overview of grid computing middleware - The Globus toolkit is presented. The key grid components that make up a grid computing environment were also described in this chapter.

Chapter 3

Related Work

This chapter presents the related work to this thesis. The background information on Grid Monitoring Architecture (GMA) and its relevance to this work is discussed in Section 3.1. A number of existing grid monitoring tools including Hawkeye, Netlogger, Network Weather Service, Ganglia, Autopilot, and Monalisa are discussed in Section 3.2. Section 3.3 provides some existing and on-going research related to this work. Finally, Section 3.4 provides a brief summary of this chapter.

3.1 Grid Monitoring Architecture (GMA)

The Grid Monitoring Architecture (GMA) is developed by Grid Monitoring Architecture Working Group (GMA-WG) [47] of the Global Grid Forum (GGF). The GMA-WG was focused on producing a high-level architecture statement of the components and interfaces needed to promote interoperability between heterogeneous monitoring systems in grid computing environments [47].

The GMA is an abstract description of the components needed to build a scalable monitoring system. The goal of the architecture is to provide a minimal specification that will support required functionality and allow interoperability. A grid monitoring system is different from a general monitoring system in the sense that it must be scalable across wide-area networks and encompass a large number of heterogeneous resources.

3.1.1 GMA Requirements for Grid Monitoring Systems

With the potential for thousands of resources at geographically distant sites and tens-of-thousands of simultaneous grid users, it is critical that data collection and distribution mechanisms scale in grid monitoring systems [12]. Performance-monitoring information produced by grid monitoring systems has these properties; fixed, often short lifetime of utility, frequent updates, and stochastic (i.e., it is frequently impossible to characterize the performance of a resource or an application component by using a single value).

Since grid monitoring systems are expected to collect and distribute performance information, they must meet the following requirements:

Low latency: The performance data must be transmitted from where it is measured to where it is needed with low latency.

High data rate: The monitoring system should be able to handle performance data that is being generated at high data rates.

Minimal measurement overhead: The measurement must not be intrusive if measurements are taken often.

Secure: The owners of the monitoring sensors may place access restrictions on the data gathered by the system. In addition, the monitoring system must ensure its own integrity and preserve the access control policies imposed by the ultimate owners of the data.

Scalable: Because there are potentially thousands of resources, services, and applications to monitor, and thousands of potential entities that would like to

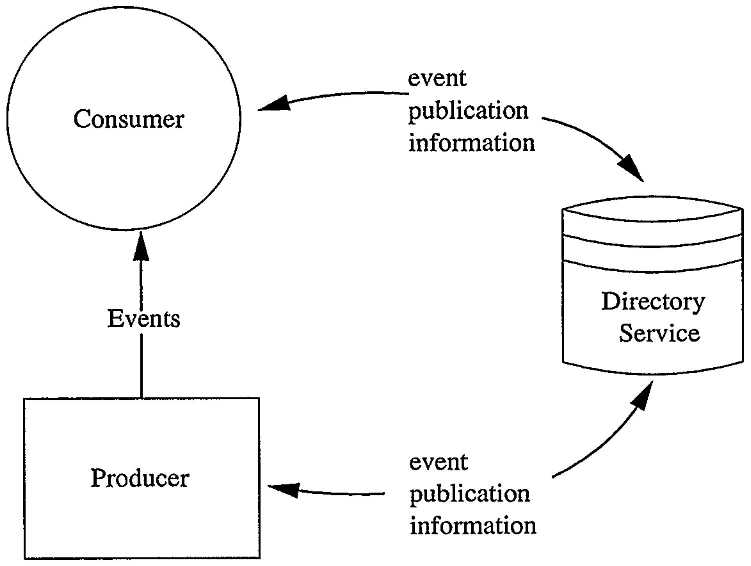


Figure 3.1: Grid Monitoring Architecture Components

receive the monitoring information, it is important that a performance monitoring system provide scalable measurement, transmission of information, and security.

In order to meet these requirements, a monitoring system must have control of the overhead and latency associated with gathering and delivering the data.

3.1.2 Terminology and Architecture

In the GMA, the basic unit of monitoring is called an event. An event is a structure containing one or more items of data that relate to one or more resources [12]. The data may relate to one or more resources such as memory or network usage or application-specific information [56].

The Grid Monitoring Architecture shown in Figure 3.1 consists of the following components:

Directory Service: Publishes what performance data are available and which producer to contact in order to request it.

Producer Makes the performance data available (i.e., performance event source).

Consumer: Requests or accepts performance data (i.e., performance event sink).

The GMA components are discussed in detail in the following sections.

3.1.3 Directory Service Interaction

Producers and consumers publish their existence in directory service entries. Consumers can use the directory service to discover producers of interest, and producers can use the directory service to discover consumers that are of interest to them.

A producer or a consumer may initiate the interaction with a discovered peer. In either case, communication of control messages and transfer of performance data occur directly between each consumer/producer pair without further involvement of the directory service [12].

3.1.4 Producer/Consumer Interactions

The GMA architecture supports three interactions for transferring data between producers and consumers: publish/subscribe, query/response, and notification. The GMA publish/subscribe interaction has three stages. In the first stage, the initiator of the interaction (i.e., a producer or consumer) contacts the "server" (i.e., the corresponding consumer or producer respectively). The purpose is to indicate interest in some set of events or data. The additional parameters needed to control the data transfer are also negotiated in this stage. These may include where to send the events, how to encode or encrypt the events, how often to send the events, buffer sizes, and timeouts. At this stage both the producer and consumer assumes a subscription state. In the next stage, the producer (i.e., the server for this interaction) sends one or more events to the consumer. In the third and final stage, either the producer or consumer terminates the subscription, possibly with additional control messages.

The GMA query/response interaction has two stages and the initiator must be a consumer. The first stage of this interaction is similar to the first stage of publish/subscribe interaction. The only difference is that the producer transfers all the performance events to the consumer in a single response after the event transfer.

The GMA notification interaction is a one-stage interaction, and the initiator must be a producer. In this interaction, the producer transfers all the performance events to a consumer in a single notification.

3.1.5 Sources of Event Data

The data used to construct events can be gathered from various sources. The sources could be hardware or software sensors that sample performance metrics in real time. Another source of event data is a database with a query interface, which can provide historical data. Also, an entire monitoring system such as the Network Weather Service [92] can serve as a source of events. Additionally, application timings from tools such as Autopilot [76] or NetLogger [25] can provide events related to a specific application.

Figure 3.2 shows one possible configuration of sources of event data. The GMA is flexible, hence it allows the performance system developers to choose any configuration that best suits their scalability and reliability needs. More information on Grid Monitoring Architecture (GMA) can be found in [12].

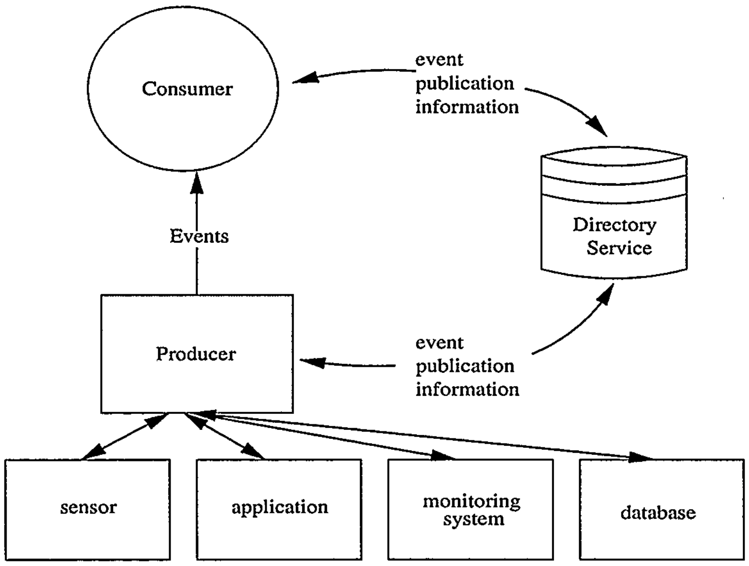


Figure 3.2: Sources of Event Data

The key components described in this thesis work are the *Monitoring tool* (i.e., the producer which makes the monitoring data available) and the *Wrapper tool* (i.e., the consumer which executes the monitoring tool).

3.2 Grid Monitoring Tools

One of the challenges in grid computing is how the quality or performance of grid infrastructure and grid applications can be measured. There is no widely accepted and deployed technique that can solve all aspects of the problem. In order to measure the performance of computing grids, numerous approaches to performance monitoring and evaluation yield different tools. Some of the tools are fully grid enabled, some are called grid monitoring tools but do not provide monitoring information related to jobs, some try to give a whole solution, and some just focus on solving a particular monitoring problem [76].

Monitoring systems, in the broadest sense, are tools that report some set of measurements to higher-level services. All monitoring systems have three major components: information collectors (sensors or probes), support services (collection, archiving, management), and interfaces (GUIs or APIs) [64]. There are tools for monitoring performance in grid computing environments; the following section describes some selected grid monitoring tools.

3.2.1 Hawkeye

Hawkeye [22] is a monitoring and management tool for clusters of computers [93]. It utilizes the technologies already present in Condor [28]. It provides rich mechanisms for collecting, storing, and using information about computers. ClassAds (Classified Advertisements) [21] are used for describing jobs, workstations, and other resources. They are exchanged by Condor processes to schedule jobs and logged to files for statistical and debugging purposes [27].

Hawkeye is based on Condor, hence the configuration of Hawkeye is extremely flexible. Hawkeye works by configuring Condor such that it periodically executes specified program(s) (typically scripts). The program produces output in the form of ClassAd attribute/value pairs, which are then added (using defined naming conventions) to the machine ClassAd. A Hawkeye system can be used to monitor various attributes of a collection of systems. The monitoring mechanism may also be used to further the management of systems but can not be used to monitor jobs in a grid computing environment.

3.2.2 Netlogger

The Network Application Logger Toolkit (NetLogger) monitors, under realistic operating conditions, the behaviour of all the elements of the application-to-application communication path in order to determine exactly what is happening within a complex system [25]. The NetLogger Toolkit has the following four features: NetLogger message format, NetLogger client library, NetLogger visualization tools, and NetLogger host and network monitoring tools. Also, NetLogger uses an additional component for synchronizing the clocks of all hosts in the distributed system. NTP (Network Time Protocol) [26] or a GPS host clock is used in the synchronization. NetLogger also includes wrappers for several system monitoring utilities, such as vmstat, iostat, and netstat [11] but it does not monitor jobs on a computing system.

3.2.3 Network Weather Service

Network Weather Service (NWS) is a distributed, generalized system for producing short-term performance forecasts based on historical performance measurements. The system dynamically characterizes and forecasts the performance deliverable at the application level from a set of network and computational resources [92]. NWS operates a distributed set of performance sensors (like CPU monitors and network monitors) from which it gathers readings of instantaneous conditions and then uses numerical models (mean-based, median based, and autoregressive methods) to generate forecasts of what the conditions will be for a given time frame [110].

The NWS uses four component processes:

Persistent State process (memory): stores and retrieves measurements from persistent storage.

Name Server process: used in binding process and data names with low-level contact information for example, TCP/IP port number.

Sensor process: gathers performance measurements from a specified resource.

Forecaster process: produces a predicted value of deliverable performance during a specified time frame for a specified resource.

In spite the fact that NWS is a good monitoring tool, it does not provide any useful information about jobs on a computer.

3.2.4 Ganglia

Ganglia [31] is a scalable distributed monitoring system for high-performance computing systems such as clusters. Ganglia has a hierarchical design targeted at federations of clusters. It relies on a multicast-based listen/announce protocol to monitor the state within clusters and uses a tree of point-to-point connections among representative cluster nodes to federate clusters and aggregate their states [78]. In Ganglia, data is represented in XML using XDR (External Data Representation Standard) [98].

Ganglia is comprised of two components, the *Gmon* (i.e., local-area monitoring system) and the *Gmeta* (i.e., wide-area monitoring system). Gmeta processes and presents the monitoring information gathered from one or more clusters running the Gmon local-area monitor. Ganglia gives monitoring information about computing systems but does not provide information that can help users to understand the behaviour of their jobs.

3.2.5 Autopilot

Autopilot [90] is a distributed performance monitoring, resource control, and tuning system that is based on the Pablo performance toolkit [84]. Autopilot is complemented by Virtue [95] - an environment that accepts real-time data from Autopilot and allows users to change software behaviour and resource policies [110]. Autopilot and Virtue allow application developers and performance analyst to capture, analyze, and steer distributed applications [104].

Autopilot is used in Grid Application Development Software (GrADS) project [30] to monitor performance contracts via application level autopilot sensors. A real-time monitor compares an application progress against the requirements of its contract and triggers corrective actions in case of violations.

The Autopilot library contains distributed performance sensors, software actuators, behavioural classification tools, Self-Defining Data Format (SDDF), decision procedures, distributed name servers, and sensor and activator clients [110]. Autopilot provides performance daemons to capture network and operating system data on distributed hosts but does not capture information about the jobs on distributed hosts in a grid computing environment.

3.2.6 Monalisa

MonaLISA (Monitoring Agents in A Large Integrated Services Architecture) system provides a distributed service for monitoring, control, and global optimization of complex systems. MonALISA is based on a scalable Dynamic Distributed Services Architecture (DDSA) implemented using Java/JINI and Web Services technologies.

MonALISA is an ensemble of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services. The services are able to collaborate and cooperate in performing a wide range of monitoring tasks in large scale distributed applications; they can be discovered and used by other services or clients that require such information. MonALISA is designed to easily integrate existing monitoring tools and procedures, and to provide this information in a dynamic and self-describing way to any other services or clients [54].

The only information that MonALISA provides about jobs on a computer is the number of queued and running jobs. It does not provide additional information on the behaviour of jobs on the computer.

3.3 Grid Monitoring Systems

This section describes some of the existing grid monitoring systems with emphasis on job monitoring.

3.3.1 Monitoring Jobs Using Mobile Agents

The proposed systems in [10, 81] use agent based technology to do monitoring in grid environments. A mobile agent is a software module that is able to migrate among the hosts of a network, in order to carry on a specific task [70]. The agent is not linked to the system where it starts its execution. After being created in an execution environment, an agent can carry its state and code to another execution environment in another host of the network, where the execution can be restarted or continued [81].

In [10], a collection of software agents [39] is used as an event management system designed for grid environments. An automated agent-based architecture called Java

Agents for Monitoring and Management (JAMM) is also developed in [10]. The implementation of the agents is based on Java and Java Remote Method Invocation (RMI). The agents can be used to launch a wide range of system and network monitoring tools, and then extract, summarize, and publish the results. The JAMM system is designed to facilitate the execution of monitoring programs, such as netstat, iostat, and vmstat, by triggering or adapting their execution based on actual client usage. JAMM is often used to collect monitoring events for use with the NetLogger Toolkit. In JAMM architecture, monitoring data is collected at both the client and server host, and at all network routers between them. Then, all event data is sent to a real-time monitor consumer for real-time visualization and NetLogger analysis. The server and router data is also sent to the archive.

In a mobile agent approach, the agent moves close to the data to be processed, thus eliminating the network traffic due to messages (excluding the initial migration), and allowing the execution of operations dynamically defined by the user [81]. In [81], it is assumed that the mobility of some code modules can contribute to the development of a more effective and flexible architecture. The advantages of using a mobile agent includes reduction of the network load, filtering of monitoring data at several abstraction levels, asynchronous and independent execution of tasks defined by a user, integration of heterogeneous resources monitoring tools, and on-demand enabling of the required services. The mobile agent system known as MAP (i.e., Mobile Agent Platform) [3] was developed at the University of Catania; it was used in [81].

The problem with the agent based approach is that it is assumed that the mobile agent can migrate to any system or cluster and run there. This is not always the case in grid environments where the resources in a grid are owned by different organizations. In a production grid it may not be possible to assign a monitoring task to an agent because a user may not have the permission to run programs on some clusters or machines.

The data collected through JAMM does not give any information about jobs and their behaviour in a grid computing environment.

3.3.2 Monitoring Jobs Using Globus Toolkit

The core services, interfaces, and protocols of Globus Toolkit allow users to access remote resources as if these resources were located within the users own machine room, while simultaneously preserving local control over who can access resources and when [37].

Kejing et al. [52] presents the architecture and implementation of a Grid Monitoring System based on Globus Toolkit, although the architecture and implementation has important practical value for the monitoring of the China Education and Research Grid (CERG). The start-up of the monitor and collection of data from remote sites are more complex and difficult than in a single cluster [10].

Globus Toolkit version 3.0, which includes a set of core services such as security, communication, managing distributed applications, and information, is responsible for the security problem brought by spanning clusters, aggregating host information from clusters and indexing host information for quick querying [52]. To get the aggregate information, one Monitoring Service runs on each organization or resource site. A large organization that consists of multiple large sites will run its own Index Service that will index the various resources available at that site.

Since the work is based on Globus Toolkit version 3.0, then there may be need to modify a grid monitoring service whenever a newer version of Globus toolkit is released. Also, the approach does not discuss how jobs can be monitored in grid environments using the Globus toolkit.

Globus only provide information on the status of a job - PENDING, ACTIVE, DONE, or FAILED. It does not provide information that can help users to understand the behaviour of their jobs. For example, the reason(s) their job failed or is in "pending" state.

3.3.3 Job Monitoring on Legion

Legion [4] is a middleware; it connects networks, workstations, supercomputers, and other computing resources into a system that can encompass different architectures, operating systems, and physical locations. A small set of attributes about jobs that are of interest to users are presented in [2]. The attributes discussed include status of a job, name of the machine on which a job is running, working directory of a job, list of files in the working directory of a job, and permissions, timestamp, and size of any file in the working directory of a job. Some grid systems already provide mechanisms for retrieving some of these attributes, whereas others do not. In [2], Legion was used as the implementation platform for demonstrating how job attributes could be retrieved.

Each system and application component in Legion is an object. The running jobs (or instances of programs) are known as objects in Legion. Legion has tools for starting a legion *runnable* object on some machine and tools for finding out the status of a running job. The work in [2] uses the inbuilt tools in Legion to monitor the status of running jobs, but only a few computing grids run Legion operating system. Most of the existing computing grids are running Globus Toolkit middleware, therefore

Legion middleware is not common.

Although, Legion provides the status of jobs on a computer, it does not provide information that could help users to understand the behaviour of their jobs on the computer.

3.3.4 Job Monitoring in Interactive Grid Analysis Environment

The design of a Job Monitoring Service is presented in [1]. The Job Monitoring Service is a web service that will provide interactive remote job monitoring by allowing users to access different attributes of a job once it has been submitted to the interactive Grid Analysis Environment (GAE) [61]. The set of interacting web services in interactive GAE include Data Collection service, Monitoring service, Execution service, and Replica Management service.

The GAE focuses on the construction of an infrastructure that allows scientists to interactively perform analysis and submit small jobs in quick succession, depending on the output of previous jobs, instead of submitting one large batch job [1]. A job monitoring service which is designed for use in a Grid Analysis Environment is being developed. The Job Monitoring Service will continuously monitor the jobs that have been submitted and whenever the state of a job changes the Job Monitoring Service will update the repository and MonALISA. The Job Monitoring Service provides an API that allow the clients to access the job monitoring information of their jobs. One of the disadvantages of this method is that it cannot be used elsewhere except in GAE.

The job monitoring service developed in GAE uses an API to provide job monitoring information such as job status, elapsed time, estimated run time, queue position, completion time, and CPU time used; but it does not provide information that could help users to understand the behaviour of their jobs on a computer.

3.3.5 Job Monitoring in GridLab

A flexible and efficient monitoring system was designed and implemented as part of GridLab project [49]. The architecture of the monitoring system is based on the GMA and exploits its distributed design, compound producer-consumer entities, and generality. This monitoring system also support advanced functions like actuators and guaranteed data delivery. Actuators are analogous to sensors, but instead of taking measurements of metrics, they implement controls that represent interactions with either the monitored entities or the monitoring system itself.

The monitoring system is not responsible for the permanent storage of monitoring data; hence, if a consumer needs to preserve monitoring data, the monitoring system must either save the data itself or supply a contact point for a storage service and appropriate credentials for authentication. The monitoring system consists of a Local Monitor (LM) service running on each node and collecting information processes (P) running on the node. The collected information is sent to a Main Monitor (MM) service. The MM is a central access point for local users (i.e., site administrators and non-grid users). Grid users can access information via the Monitoring Service (MS) which is also a client of MM [13]. The Local Resource Management System (LRMS) controls jobs running on hosts belonging to a grid resource. It allocates hosts to jobs, starts and stops jobs on user request, and possibly restart jobs in case of an error. This approach requires processes to register and identify themselves to the monitoring system at startup.

The monitoring system relies on the application to honestly register all its processes; hence, a faulty application would prevent correct monitoring because it will generate incorrect data [13]. The proposed solution is to submit jobs via the *jobmanager* (e.g., Globus GRAM [69]). The problem with this system is that it is assumed that Globus is running on all the computers in a grid environment but this may not be true. In addition, only few users do submit their jobs via Globus; many users still prefer to submit their jobs directly to the batch scheduler. Therefore, jobs not submitted via Globus cannot be monitored because the monitoring system relies on Globus GRAM.

The monitoring system in GridLab collect information about applications but this information is not useful in understanding the behaviour of jobs on a computer.

3.3.6 Monitoring Grid Applications

Grid applications access distributed and often shared computing resources. One consequence of this resource sharing is that measured application performance can vary widely in unexpected ways [29]. Determining the causes of poor performance due to either anomalous application behaviour or contention for shared resource use, and adapting to changing circumstances are critical to creation of robust grid applications.

An infrastructure for grid application contract development and monitoring is described in [29]. This infrastructure is based on the Autopilot toolkit, and provides flexible and scalable tools to assess both application and system behaviour. Performance contracts and real-time adaptive control were the mechanisms used to realize soft performance guarantees in grid environments. Performance contracts formalize the relationship between application performance needs and resource capabilities. During execution, contract monitors use performance data to verify that expectations are met. When the contracted specifications are not satisfied, the system can choose to either adapt the application to available resources or reschedule the application on a new set of resources that can satisfy the original contract specifications.

The application monitoring infrastructure described in [9] is developed within the CrossGrid project [24]. The monitoring environment is composed of a distributed monitoring system, the OCM-G [8], and a performance analysis tool, G-PM (Grid-oriented Performance Measurement). The purpose of the environment is to collect data about running applications and enable the user to observe their performance in on-line mode, so the user can dynamically change measurements to support and solve performance problems [9].

SCALEA-G is implemented based on the Open Grid Services Architecture (OGSA). It provides an infrastructure for conducting online monitoring and performance analysis of a variety of grid services including computational and network resources, and grid applications [53]. Source code and dynamic instrumentation are implemented to perform profiling and monitoring in SCALEA-G.

The work in [79] combines GRM [111] application monitoring tool, and the Mercury resource and job monitoring infrastructure to provide an on-line grid performance monitoring tool-set for monitoring message-passing parallel applications executed on a grid.

Grid monitoring systems discussed in this section focus on monitoring grid applications for tuning and debugging. They do not have the capability to monitor jobs on a computer. Also, they do not provide information that could help in understanding the behaviour of jobs.

3.3.7 Monitoring using Asynchronous Middleware - DREAM

DREAM (Dynamic REflective Asynchronous Middleware) is introduced in [102]. DREAM is a Java component-based message oriented middleware. Asynchronous communications are used to achieve the scalability and flexibility objectives, whereas the reflective component technology provides the complementary configurability and adaptability features. DREAM provides a JMS (Java Message Service) [99] implementation, thus making monitored data accessible to J2EE application servers. The monitoring service is integrated with the Open Grid Software Architecture (OGSA [57]). DREAM has similarities with JAMM [10] architecture; JAMM is an agent-based system that automates the execution of monitoring sensors and the collection of data events.

Globus Toolkit is the de facto middleware for grid computing; it is difficult to convince organizations to use DREAM middleware in order to monitor grid resources. This approach is based on J2EE application servers which may not be available on some computers. The shortcoming of this system is that the monitoring service in DREAM does not monitor jobs on a computer.

3.3.8 Job Centric Cluster Monitoring System

The Job Centric Monitoring system is designed to monitor jobs across a cluster in grid environments. This approach is being developed in Grid Research Centre at the University of Calgary. Sequential jobs and parallel jobs can be monitored using this methodology [74, 87].

In this approach, a job monitoring tool is deployed on all the computers in the computing environment and an application monitoring tool is deployed on a computer that could pull information from other computers. The job monitoring tools on the individual computers are started whenever the application monitoring tool is executed. The application monitoring tool returns a large amount of monitoring data which includes user ID, process ID, node name, and other information about the processes on a particular computer.

The *username* of a user could also be used as criteria for extracting the processes that are currently running on a computer and belongs to the user. The job monitoring tool on each computer forwards the monitoring data of all the processes that are currently running on the computer to the application monitoring tool. The application monitoring tool receives the monitoring data from several computers and filters the received data. The useful monitoring information is kept while others are discarded.

This approach makes it possible to track and monitor parallel jobs. This method of monitoring jobs is suitable for monitoring jobs on a large scale. For example, all the jobs on all the nodes of a cluster computer. This system is cumbersome for monitoring a single job, because the system requires a number of tools to be installed on the grid cluster. The method developed in this thesis uses a simple script to monitor an individual job on a computer.

3.4 Summary

The Grid Monitoring Architecture which describes the components and interfaces needed to promote interoperability between heterogeneous monitoring systems in grid computing environments was presented. Most of the existing grid monitoring tools that are relevant to this thesis work were described in this Chapter. Some of the selected monitoring tools include Hawkeye, Netlogger, Network Weather Service, Ganglia, and Monalisa.

Some existing work related to this thesis are described in this Chapter. The reasons why there is need for a monitoring system that can monitor individual jobs were identified.

Chapter 4

Job Monitoring

This chapter focuses on the design and the implementation of a technique for monitoring jobs in grid computing environments called the *Wrapper method.* This technique would help users to understand the reason(s) their jobs behaved in a certain way on a computer.

How the *Wrapper method* works is described in detail in Section 4.1. Section 4.2 highlights the important information that users would like to know about their jobs. The sources of job monitoring data are described in Section 4.3. Section 4.4 describes how the monitoring information is moved to a location accessible by users. The method by which the monitoring information is presented to the users is described in Section 4.5.

How the *Wrapper method* is implemented in this thesis work is described in Section 4.6. Section 4.7 discusses the issues with the specific design of the *Wrapper method* in this thesis. How the *Wrapper method* is used in a grid computing environment with Globus middleware is described in Section 4.8. Finally, this chapter is summarized in Section 4.9.

4.1 Proposed Methodology - Wrapper Method

This section describes a proposed technique for monitoring jobs in grid computing environments. The technique described in this section is named the *Wrapper method* because a script known as the *Wrapper script* is used in the process of monitoring a job. The word Wrapper is used because the script is used to wrap a job; therefore, the script is submitted in place of the job. The Wrapper script starts and monitors a job on whatever computer the job is scheduled to run. Figure 4.1 shows how the *Wrapper method* works.

In this method, the Wrapper script is submitted to the batch system and the batch system executes the Wrapper script. When the Wrapper script is started on a computer, the job is executed and a monitoring tool is started on the same computer to monitor the job.

When the Wrapper script is executed, it *forks* a new process as shown in Figure 4.1. The newly created process is the *child process* and the existing process (i.e., the Wrapper script process) that created the new process is the *parent process.* The child process is used to execute the job while the parent process is used to monitor the job. When the *fork* system call is made, the *process ID* of the child process is returned in the parent and a value of 0 is returned in the child process. Monitoring the job in the parent process makes it possible for the parent process to wait for all its children before it exits. In other words, job monitoring is carried out in the parent process because the monitoring process will not exit until the monitored process no longer exists.

The job is started using the *exec* system call within the child process. A successful *exec* system call does not return any value because the calling process is replaced by the new process. That is, when the job is executed in the child process, the child process is completely replaced by the job. The monitoring process is not intrusive; it does not affect the performance of the job.

The job monitoring is initiated within the parent process and continues until

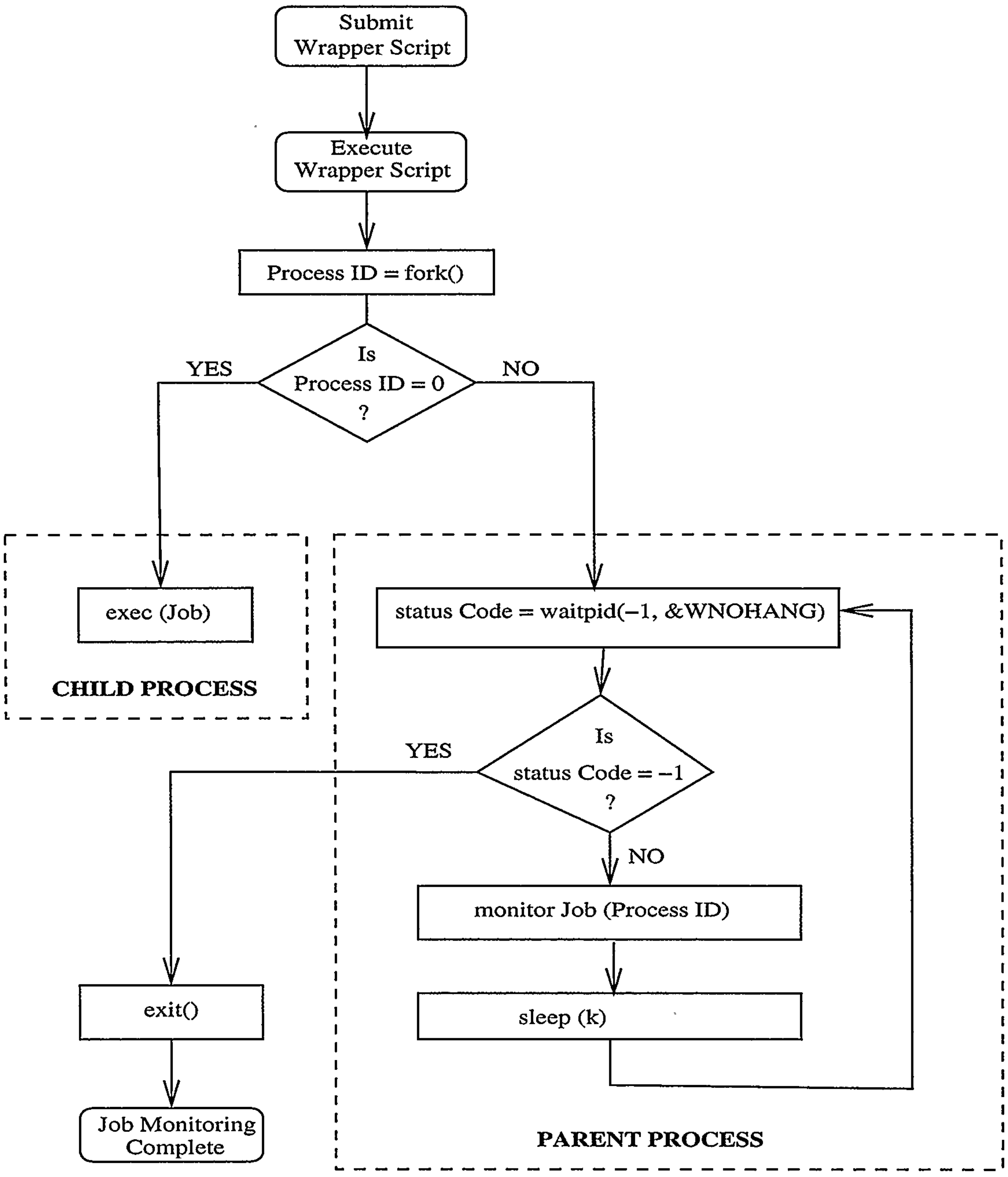


Figure 4.1: Wrapper Methodology

the job is completed. In order to monitor the job that is being executed in the child process, the *waitpid* function is used to suspend the execution of the current process until the child process or the children processes have exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function [97].

Within the parent process, the *waitpid* function takes in two arguments; the arguments specify the process(es) the parent process should wait for, when and the condition under which *waitpid* should return. The use of constant values of *WNOHANG* and *-1* in the *waitpid* function causes the parent (monitoring) process to wait for the child process and any of its children. These options make it possible for the monitoring tool to track all the processes that are associated with a job. In addition, these options allow the monitoring to be carried out uninterrupted while the job is being executed in the child process. It is important to monitor the behaviour of processes that are *forked* within a job because they are associated with the job and are part of the job.

The return value from *waitpid* is checked each time it is called. A return value of -1 indicates that the child process being monitored is no longer available. Hence, the parent is terminated using the *exit* system call. The *exit* call terminates the parent process and returns an exit status code to the calling environment. An exit status code of 0 indicates a successful operation and any other value indicates a problem.

An exit status code of any value other than -1 from the *waitpid* indicates that the job is still running. If the job is still running, the monitoring system retrieves monitoring data that is related to the job using the *process ID* of the job. These steps are carried out over and over again until the job is completed and *waitpid* returns a value of -1. A delay is introduced by calling the *sleep* function with a value of k seconds within the parent process; the value of k determines how often the job is monitored.

The monitoring stage of the *Wrapper method* involves retrieving performance statistics related to a job from a computer where the job is running. The *process ID* of the job is used in retrieving information that is specific to a particular job. The monitoring information include page fault, CPU, and memory statistics related to the job. The first step in the monitoring stage is to check whether the job has other processes (i.e., children processes) related to it. The monitoring information for the job and its children are retrieved from the computer, processed, and stored in computer data files. In addition, the monitoring system logs information that is related to shared resources activity on the computer. The shared resources considered are file system, disk Input/Output (I/O), and network.

During this stage, the monitoring system logs the memory usage statistics of other processes that are not related to the job on the computer. This information helps in detecting competition for system resources between the job and other unrelated tasks on the same computer. The time at which the job was monitored is logged alongside the monitoring data. The memory usage statistics of the host computer is collected whenever the job is monitored.

This method can be used to monitor sequential jobs and shared-memory parallel jobs on a single computer. The problem with this method is that it cannot be used to monitor a distributed-memory parallel job. This problem is due to the fact that a scheduler usually does not know how a parallel job is started. Therefore, the monitoring tool will lose track of some processes that belong to the job since it does not know how to relate the processes to the job. It is possible to know the hosts on which the processes associated with a parallel job are being executed, but it is difficult to know exactly what processes belong to the parallel job in a situation where the user has other processes running on the same computer at the same time. This will be considered as future work and it is discussed further in Chapter 6.

4.2 Job Monitoring Data

This section discusses the monitoring information that is collected by the job monitoring tool developed in this thesis. The information in the logs of batch schedulers does not give details of the resources used by a job. The accuracy of the data in the logs of batch schedulers is not guaranteed. In order to understand the behaviour of a job on a computer, there is need to consider the usage statistics of the resources used by the job.

The monitoring statistics collected about a job on a computer by the monitoring tool in the *Wrapper method* includes CPU utilization, memory usage, network bandwidth utilization, file system activity, and disk I/O activity. Some existing tools provide some of this information but not in time-series format. The monitoring tool developed in this thesis work provides time-series information about resources used by a job.

The importance of monitoring statistics is highlighted in the following sections. The monitoring information considered in this work is described with respect to Linux operating system, although the description of the monitoring data is related to other operating systems.

4.2.1 CPU Utilization

CPU time is the amount of time used in processing a computer program on a CPU; and CPU utilization is the program's share of the total CPU time. CPU utilization is expressed as a percentage of the total CPU time used by a process in a single processor environment, although some operating systems represent CPU utilization in a different way. System load is the amount of work that a computer is doing. System load is usually the first factor that is considered when a job is not performing well on a computer. Some common CPU utilization statistics are described in Appendix B.

Some compute-intensive jobs (i.e., jobs that uses large amount of computing resources) have high CPU utilization. Hence, a compute-intensive job with low CPU utilization may indicate that there are processes on the system competing with the job. If there are no processes competing with the job, then this may indicate some bug or some failure in the application.

A job that is expected to use a large amount of CPU time may get less CPU time even when there is no competition from the other processes on the system. This situation may not be due to a competition for CPU time, but could be due to competition for shared resources with other jobs on the computer. Therefore, the job would not be using the CPU most of the time but waiting for I/O.

4.2.2 Memory

The computer memory is the place where the computer holds current programs and data that is in use. The physical memory is used as storage for both data and programs while a program is running on a computer. An address space refers to a range of virtual addresses accessible to or reserved for a process. Without virtual memory the amount of available physical memory has to be equal to or greater than the address space of the application to be run; otherwise, the application would fail with an "out of memory" error [88]. The operating system keeps track of which parts of memory are in use and which parts are not in use, in order to allocate memory to processes when they need it and de-allocate it when they are done.

The idea behind virtual memory is that the combined size of a program, its data, and stack may exceed the amount of physical memory available for it. The operating system keeps some part of a program currently in use in main memory and the rest on disk. The virtual address space is divided into units called pages; transfers between RAM and disk are in units of a page.

Virtual memory increases the available memory on a computer by using some additional space on the hard disk. The lack of enough memory resources may have a significant effect on the overall performance of a computer [36]. Appendix B lists the commonly found memory management statistics.

The *resident set size* is the size of the memory-resident pages in the address space of a process. The *resident set size* of a job may indicate that there is competition for physical memory if its resident set size is smaller than its virtual memory and all the physical memory on the computer has been used. The *resident set size* of a job may also indicate shortage of physical memory on a computer.

The difference between the virtual memory and resident set size used by a job would help in understanding the shortage of memory. That is, the relationship between the virtual memory and the physical memory used would help in understanding the behaviour of a job on a computer.

A program manages its virtual memory in such a way that any memory released by the program is not given back to the operating system but kept and re-used by the program. The released memory pages are added to a list of free memory pages for later use. Therefore, when additional memory is needed by a program it would check its memory pool to see if there is enough memory to handle the request, else it would request for more memory from the operating system.

4.2.3 Page Faults

A *virtual address space* is a non-contiguous memory that is presented to a process as a contiguous memory. Virtual memory describes the total number of uniquely-addressable memory locations available to a program, and not the amount of physical memory that must be dedicated to the program at any given time [88].

In order to implement virtual memory, the Memory Management Unit (MMU) manages the virtual address space and each virtual memory address goes through a translation step prior to each memory access. In order to reduce the overhead of individually tracking the virtual to physical memory address translation, RAM is divided into pages (i.e., contiguous sections of memory of a set size that are handled by the MMU as single entities). A *page fault* occurs when the MMU has no translation in cache for a memory address requested by the CPU. Therefore, the MMU interrupts the CPU and causes the page fault handler in the operating system to be executed. The page fault handler then determines what needs to be done to resolve the page fault. The operating system will resolve the problem by using free or reusable pages. In some cases, the operating system will *page out* existing pages that are not in use and write them to disk.

Page faulting may not be an issue if it does not occur in excess. Paging makes virtual memory possible by allowing the memory requirements of a process to exceed the actual amount of physical memory. On many modern systems, a program execution always starts with a page fault as the operating system tries to use the kernel's virtual memory management facility to read enough of the executable image to get it started [36]. Virtual memory makes it possible for computers to handle larger and complex programs.

A large amount of page faults and swapping activity may indicate lack of physical memory on a computer. This may lead to low CPU utilization (because the CPU cycles would be used for supporting memory management and setting up the necessary disk I/O operations) and/or high network activity if the disk volume is shared across a network. Low physical memory, heavy page faulting activity, and a system running near its limit in terms of CPU or disk I/O may lead to *thrashing* which decreases system performance rapidly. Thrashing is a situation where a computer is spending more time doing paging instead of actual work.

There are two types of page faults - major and minor page faults. Based on the definitions of major and minor page faults in the man page for the */proc* file system on Linux systems [89], a minor page fault indicates that the fault does not require loading a memory page from disk. This occurs when an attempt is made to access a virtual memory location that resides within a segment and the page is in the physical memory; but no current MMU translation is established from the physical page to the address space that caused the fault [109]. In this case, the physical page of memory is already present but the process needs to establish a mapping to the existing physical page.

A major page fault requires loading a *paged out* memory page from disk into the physical memory. That is, the page is not loaded in memory at the time the fault is generated. Hence, major page faults are more expensive than minor page faults because it adds disk latency to the execution of the interrupted program [108].

4.2.4 Network

In the context of this thesis, a network is the connection between two computers. A number of network performance metrics determines the "speed" of a network. The common network performance metrics are bandwidth, latency/delay, jitter, and reliability. Network bandwidth is the amount of data that can be sent over a network connection or interface in a given period of time. Bandwidth is usually stated in bits per second (bps), kilobits per second (kbps), or megabits per second (mbps).

Network bandwidth and latency both determine the "speed" of a network perceived by a user. Latency refers to delay in processing network data. Another factor that affects network bandwidth is *jitter*. Jitter is the variation in the time between packets arriving. It may be caused by network congestion or network route change. Some of the more common bandwidth-related statistics are listed in Appendix B.

Good network performance always depends on network devices working properly and efficiently. Network statistics like bandwidth, latency, transmit error rate, and receive error rate may indicate failing or misconfigured hardware. For example, a misconfigured network interface would cause a high transmit/receive error rates and high latency. Network adapters, routers, switches, and devices may fail and produce high error rates or degrading performance over time or both [36].

Poor network performance may be the result of too many requests to computers on a network. Overloading may be caused by too much traffic on the server network interface and/or incorrect configuration. Insufficient network bandwidth for the resources and applications on the network can also lead to poor network performance.

In turn, poor network performance would affect the performance of a job that depends on network resources; a remote I/O job for example, depends on the network for moving data from a source to a destination.

4.2.5 Disk Input/Output Activity

The UNIX operating system supports disk/system-based I/O operations in which a process interacts with a physical device using an intermediary kernel buffer. This intermediary buffer is transparent to the user's calls like *read()*, *write()*, and *lseek()*, as if they are accessing a physical file directly [71].

In disk I/O, the process interacts with a physical device directly, without the kernel's intervention. An example of disk I/O is in data-critical applications, where the user wants to ensure that the data is written to a disk immediately so that it is not lost in the event of a system failure.

Disk I/O can have an effect on the performance of a computer or the behaviour of a job. In order to understand disk I/O performance statistics, it is important to understand the disk I/O activities performed by the applications or typical workload on a computer. A large disk I/O operation can generate many pending disk I/O requests, and users needing disk volume access can be forced to wait for them to complete. A large disk I/O operation can degrade the performance of interactive activities on the same computer. For example, an appreciable waiting time would be noticed when *ls* command is issued at the commandline when a job is performing a large number of disk I/O operations on the same computer.

In order to understand some disk I/O issues, there may be a need to look at the amount of data transferred (read or written) to a local or remote disk volume during the measured interval and the amount of transfers per second.

4.2.6 File Systems

A file system is a structure that is used for storing and organizing computer files and the data they contain to make it easy to find and access them. A file system is used by a job running on a computer for reading and writing data from and to storage devices. It may use a secondary storage device such as hard disk or tape. It maintains the physical location of the files which may be virtual and exist only as an access method for virtual data or for data over a network (e.g., NFS) [107].

A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, NTFS, HFS, ext2, and ext3. Network File System (NFS) (i.e., a distributed file system developed by Sun Microsystems) is used in the experimental environment of this thesis work. NFS allows a computer to access files over a network as if they were on the local disk of the computer.

NFS-specific traffic and performance are monitored using the *nfsstat* command. This command displays statistical information about the NFS and the RPC (Remote Procedure Call) interface to the kernel for NFS clients and servers. The statistics on the types of NFS operations performed, along with error information and performance indicators would help in identifying potential bottlenecks.

Higher latency would be observed when writing or reading a large file on a remote computer through NFS than performing the same activity on a local disk. An NFS configuration problem may make the accessing, writing, or reading of a file slow. In some cases, NFS may be affected by the state of the network traffic since it has to work through the network.

Since NFS is a shared resource, the performance that is seen on a single computer locally may not be enough to understand the behaviour of a file system. This is due to the fact that the file system may be in use by other jobs running on other computers sharing the file system.

4.2.7 State of a Job

A process is an instance of a computer program running on a computer. A job or task on UNIX systems, may be composed of one or more processes working together to perform a specific task [36]. The life cycle of a UNIX process is described in Appendix C. A process can be in any of the following 3 common states:

Running: if the process is executing on a processor.

Ready: if the process could execute on a processor if one were available.

Blocked: if the process is waiting for some event to happen (for example, disk I/O completion event) before it can proceed.

The process states of a job may show why a job behaved in a certain way. For example, a job that spent much time in the blocked state indicates that the job may be doing lots of I/O; this in turn may help in understanding why a job used more or less CPU and memory resources.

4.3 Sources of Monitoring Data

This section describes the sources of the monitoring data discussed in the previous section. The description of the sources of the monitoring data in this thesis work is based on Linux operating system.

There are two main flavours of UNIX; Berkeley Software Distribution (BSD) and SysV (System V). System V was originally developed by AT&T and first released in 1983. BSD is the UNIX derivative distributed by the University of California, Berkeley starting in the 1970s. Linux has a combination of both BSD and System V flavours of UNIX. The following sections highlight the sources of monitoring data.

4.3.1 Process Information Pseudo-Filesystem

The UNIX Process Information Pseudo-Filesystem is also known as *proc*. The proc file system is used as an interface to kernel data structures. The files in the */proc* directory correspond to active processes (entries in the kernel process table). Most of the files in this directory are read-only, but some files allow kernel variables to be changed [103].

The proc directory contains information on all the processes that are currently running on the computer. The *proc* file system grants access to information about processes and other operating system features. In Linux, the monitoring information belonging to a process is stored in /proc/[number] where number is the *process ID* of the process. The description of */proc* directory in this section is based on the information in Linux Programmer's Manual. The Linux operating system contains the following pseudo-fields and directories:

/proc/[number]/cmdline: This holds the complete command line for the process, unless the whole process has been swapped out, or unless the process is a zombie. In either of these later cases, there is nothing in this file.

/proc/[number]/cwd: This is a link to the current working directory of the process.

/proc/[number]/exe: Under Linux 2.2 and 2.4, exe is a symbolic link containing the actual path name of the executed command. The exe symbolic link can be dereferenced normally - attempting to open exe will open the executable.

/proc/[number]/fd: This is a subdirectory containing one entry for each file which the process has opened, named by its file descriptor which is a symbolic link to the actual file.

/proc/[number]/maps: A file containing the currently mapped memory regions and their access permissions.

/proc/[number]/mem: The pages of a process's memory can be accessed via the *mem* file.

/proc/[number]/root: UNIX and Linux support the idea of a per-process root of the filesystem. This file is a symbolic link that points to the process's root directory, and behaves like *exe*.

/proc/[number]/stat: Gives the status information about the process. This is used by the *ps* tool described in the next section. The information found in this file includes process ID, file name of the executable, state of the process, and virtual memory size in bytes.

/proc/[number]/statm: Provides information about memory status in pages.

/proc/[number]/status: Provides much of the information in /proc/[number]/stat and /proc/[number]/statm in a format that is easier for humans to parse.

The advantage that *proc* has over other sources of monitoring information is that it contains detailed information about the process that is being monitored. One of the problems with *proc* is that it is not portable, in the sense that the structure of *proc* directory and the layout of data in some files are different from one variant of UNIX to another. Therefore, there would be need to have different implementations of the job monitoring tool for different variants of UNIX.

4.3.2 Process Status Utility

ps stands for Process Status. *ps* gives a snapshot of the current processes running on a system. The utility produces a report summarizing execution statistics for current processes. The command's options control which processes are listed and what information is displayed about each process. The format of the commands differs considerably between the BSD and System V forms. The output of the *ps* command contains the following information:

User: Name of the owner of the process.

Process ID: The process ID of the process that is currently executing.

CPU Percentage: This shows the CPU time or real time percentage. It is the process' share of the CPU time expressed as a percentage of the total CPU time per processor.

Size: The total size of the process in virtual memory, including all mapped files and devices, in kilobyte units.

Virtual Memory: The total virtual memory size in bytes.

TTY: The terminal associated with the process.

Start: The starting time of the process, given in hours, minutes, and seconds. The start time for a process that began more than 24 hours before the *ps* inquiry is executed is given in months and days.

Time: CPU time used by the process since it started.

Command: The simple name of the executable.

Resident set size: Real memory (resident set) size of the process, in kilobyte units.

Memory Utilization: The ratio of the process resident set size to the physical memory on the machine, expressed as a percentage.

Process State: This indicates the current state of the process. The process may be in any of the following states:

- D - Uninterruptible sleep (usually I/O)
- R - Running or *runnable* (on run queue)
- S - Interruptible sleep (waiting for an event to complete)
- T - Stopped, either by a job control signal or because it is being traced
- W - Paging (not valid since the 2.6.xx kernel)
- X - Dead (should never be seen)
- Z - Defunct ("zombie") process, terminated but not terminated by its parent

For BSD formats and when the stat keyword is used, additional characters may be displayed:

- < - High-priority (not nice to other users)

- N - Low-priority (nice to other users)
- L - Has pages locked into memory (for real-time and custom IO)
- s - Is a session leader
- l - Is multi-threaded
- + - Is in the foreground process group

The advantage of using *ps* is that its output is formatted, so it makes it easy to extract the statistics about a particular process. The monitoring information from *ps* is displayed as output (at commandline or written to a file) when it is executed. In addition, the output contains information about all the processes that are currently running on the computer. Hence, it makes it possible to know and distinguish the processes that may be competing with another process. All the variants of UNIX operating system have *ps* utility and their outputs are similar for most variants of UNIX.

4.3.3 System Tools

Job monitoring data could be retrieved from a system using various UNIX tools. UNIX system monitoring tools and commands can be grouped into the following categories:

- CPU Utilization: *top* display information that is related to process(es) associated with a job including CPU utilization. The other CPU activity monitoring tools are *uptime* and *sar*.
- Memory: *vmstat*, *free*, and *sar* could be used to gather memory usage statistics.

- Network examination: *netstat*, *ping*, *traceroute*, *ntop*, and *tcpdump* can be used to monitor impact of jobs on the network traffic.
- System Devices: *iostat*, *vmstat*, and *sar*
- Input/Output operations: *lsof*
- Network File System activities: *nfsstat*

The information derived from the use of the afore-mentioned system monitoring tools is listed in Appendix D. These monitoring tools are powerful, but none of these tools produces all the needed monitoring information. Most of the monitoring data collected in this thesis work, is retrieved from the computer using the *ps* tool.

4.4 Collecting Monitoring Data

The monitoring data in its raw form contains values of monitoring statistics retrieved from the host computer where the job was executed. The monitoring data is moved from the host computer to a location where it is processed further into meaningful monitoring information.

There are several ways of moving monitoring data from the execution host to another location. Some users may prefer to leave the data at the location where the job was submitted; others may prefer to transfer the monitoring data to another location. In the second case, a user can use any remote file transfer utility like *Secure Copy* independently or through the batch system. When a batch system is used, the monitoring data is moved together with the output of the user's job. The process of moving a file or files off the execution host after the job completes execution is known as *stage-out*.

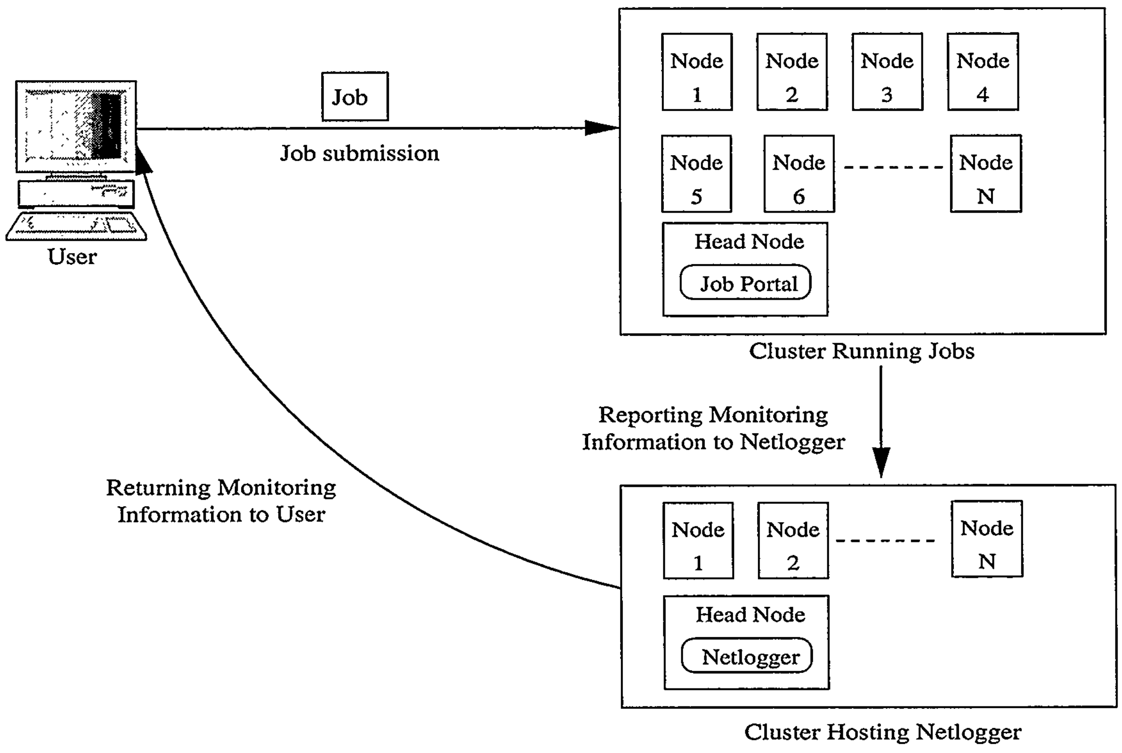


Figure 4.2: Logging Monitoring Data through Netlogger

The monitoring data could also be collected through Netlogger instrumentation. The Netlogger toolkit makes it possible for distributed applications to log interesting events at various points in the applications. Netlogger has been used for developing tools for host and network monitoring. In this approach, the monitoring tool is instrumented with Netlogger so that the monitoring information is transferred via an open system port from the system on which the job is running to a computer that is running Netlogger.

In order to use Netlogger to log monitoring data, all the monitoring events must use a common logging format, common set of attributes, and a globally synchronized timestamp. When the job is monitored at a given point in time, all the monitoring data are logged and collected at a central location.

The advantage of using Netlogger is that the monitoring data is not stored on the local system but transferred to the central location during monitoring. Hence, the monitoring data can be accessed while the job is running instead of waiting until the job is completed before accessing the data. It also makes it possible to monitor jobs in real time since the user can have access to the monitoring information on the head node or a remote computer while the job is still running.

There are several ways of returning monitoring data to users. Some of the methods are simple while others are complicated. The objective of the *Wrapper method* is to use an approach that is as simple as possible.

4.5 Presenting Monitoring Information

In order to make sense out of the monitoring data, there is need to transform the monitoring data into meaningful information that can be easily understood by users. The monitoring data is transformed into graphical format. The monitoring information graphs include CPU utilization, memory usage, page faults, network traffic, file system, and disk I/O operations.

The monitoring information is presented in time-series format in order to show the complete life cycle of a job. The monitoring information can be presented to users via a job portal. The job portal will display the monitoring information that is specific to a user's job. In this method, the user can connect to the job portal from any system and the user would have access to the monitoring information related to his/her job. All the user need in order to access the monitoring information is a Web browser and an Internet connection.

A Web server that is accessible by users can be setup on a computer to host the job portal. The address of the Web server and the port number (if it's different from the default http port) are given to the users, so they could connect to the job portal from anywhere at anytime. The advantage of using a job portal is that the user can connect to the job portal using any standard browser, hence there is no need to write, build, install, and/or configure another application in order to monitor the progress of jobs in post-mortem.

4.6 The Implementation of Wrapper Method

How the *Wrapper method* is implemented in this thesis work is described in this section. This section describes how the job monitoring is carried out, how the monitoring data is collected, transformed, and presented to users.

The *Wrapper script* used in the *Wrapper method* is written in *Perl (Practical Extraction Report Language)* in Linux environment. Perl is used because it has easy but powerful text manipulation features. The *Wrapper script* starts the job and continues to monitor the job until it is completed as described in Section 1 of this chapter.

Whenever the job is monitored in the parent process, the monitoring tool checks the status of the child process using *waitpid* to see if the lead process (i.e., the job) or any of its children are still running. If any of them was running, the job and its children are monitored by starting the monitoring tool with the *process ID* of the job. The monitoring tool executes the *pstree* command with the *process ID* of the job in order to retrieve all the processes that are associated with the job. This is important because the job's lead process and its children are all part of the job that is being monitored. Hence, the monitoring information about the lead process and its children would help in understanding the behaviour of the job.

During the monitoring activity, *ps aux* is executed on the host computer. The output of *ps aux* is passed as input to the regular expression parser (egrep) for the lead process and its children. The *egrep* command extracts the output associated with each process from the initial output. The information retrieved includes CPU utilization and memory usage statistics. In addition, disk Input/Output, file system, and network statistics are collected on the host computer. The significance of monitoring data was discussed in Section 4.2.

The monitoring information collected about a job is recorded in files using the current system time as timestamp. The use of system monitoring time helps in understanding why a job behaved in a certain way during its life cycle.

In this thesis, the monitoring data collected is written to files in a location specified in the job submission script. If the user does not specify a location, the monitoring data is stored in the user's *working directory* (i.e., where the job was submitted). Then, the monitoring data is processed by a graphing tool developed using *Perl* and *Gnuplot.* The graphing tool generates monitoring information graphs from the monitoring data.

4.7 Implementation Issues

The time when users could have access to the monitoring information about their jobs may be a concern for some users. Some users may prefer to see the way their job is performing in real time, so they could notify the system administrator if their job is not doing well on a computer.

A "post-mortem" approach is used in presenting the monitoring information to users in this implementation. As the name implies, the monitoring information is transferred to a location specified by the user, after the job is completed. Here, the user only has the opportunity to know what happened to his/her job after it is completed. In this implementation, a user cannot have access to the monitoring information until the job is finished. An assumption is made that most users would like to know what happened to their jobs after they are completed.

4.8 Wrapper Method Implementation with Globus

Since most grid computing sites use Globus middleware, it is important to show that the *Wrapper method* can work with Globus. That is, jobs submitted via Globus can be monitored using the *Wrapper method*. The Globus Toolkit includes tools for authentication, scheduling, file transfer, and resource description. In addition, it provides the opportunity for users to submit their jobs remotely from their local computer. The implementation of *Wrapper method* in this thesis was carried out in a cluster environment with one scheduler and a batch system.

In order for users to submit their jobs in a grid computing environment running Globus Toolkit, there is need for authentication. The Grid Security Infrastructure (GSI) provides the authentication and authorization mechanisms to verify a user-supplied "Grid credential". To sign-on once to a grid, a user needs to create a temporary credential called a proxy certificate. The proxy certificate confirms that the user is authorized by a trusted authority to access the grid resources. It confirms the user with the *passphrase* he/she used in creating his/her X.509 certificate and key. Once a proxy has been created using *grid-proxy-init* command, the user can submit his/her job.

The Globus Grid Resource Allocation Manager (GRAM) framework provides services for submitting, monitoring, and cancelling jobs on grid computing resources. The framework does so through the use of the Resource Specification Language (RSL), which provides a simple set of directives for specifying typical computational parameters, such as the number of nodes and processors required, the length of time the job should run, and the executable that should be launched [60].

With Globus middleware, the *Wrapper script* is submitted to a computing grid using specific Globus commands: *globus-job-submit*, *globus-job-run*, or *globusrun*. A user would submit a RSL description of his/her compute task to GRAM running on a given resource. GRAM is a root-level process which handles all globus job requests at a remote site. When a job is submitted, the request is sent to the Gatekeeper of the remote computer. The Gatekeeper handles the allocation request and creates a Job Manager for the job. The Job Manager starts and monitors the remote program, communicating state changes back to the user on the local machine [101].

After the authentication, the Globus GateKeeper starts the requested service, in this case the Job Manager. The Job Manager process translates the generic RSL parameters into a job script matching the local resource manager and submits the job script. The Job Manager keeps track of and manages grid I/O for jobs running on the local batch system. There is a specific Job Manager for each type of batch system supported by Globus (examples are Condor, LSF, LoadLeveler, and PBS) [62] [63].

A local job scheduler is required in order to manage the resources of the compute element. GRAM has the ability to spawn simple time-sharing jobs using standard

UNIX *fork* methods, but most large-scale compute elements will be under the control of a scheduler such as PBS, LSF, Condor, NQE, or Loadleveler [100]. The description of remote execution of jobs on Globus-managed grid computing environment is shown in Figure 4.3.

When the Wrapper script is executed on a local host, the job embedded within the Wrapper script is executed and monitored using the *Wrapper method* described in Section 4.1. The monitoring process in a grid computing environment with Globus middleware is the same as in a cluster environment.

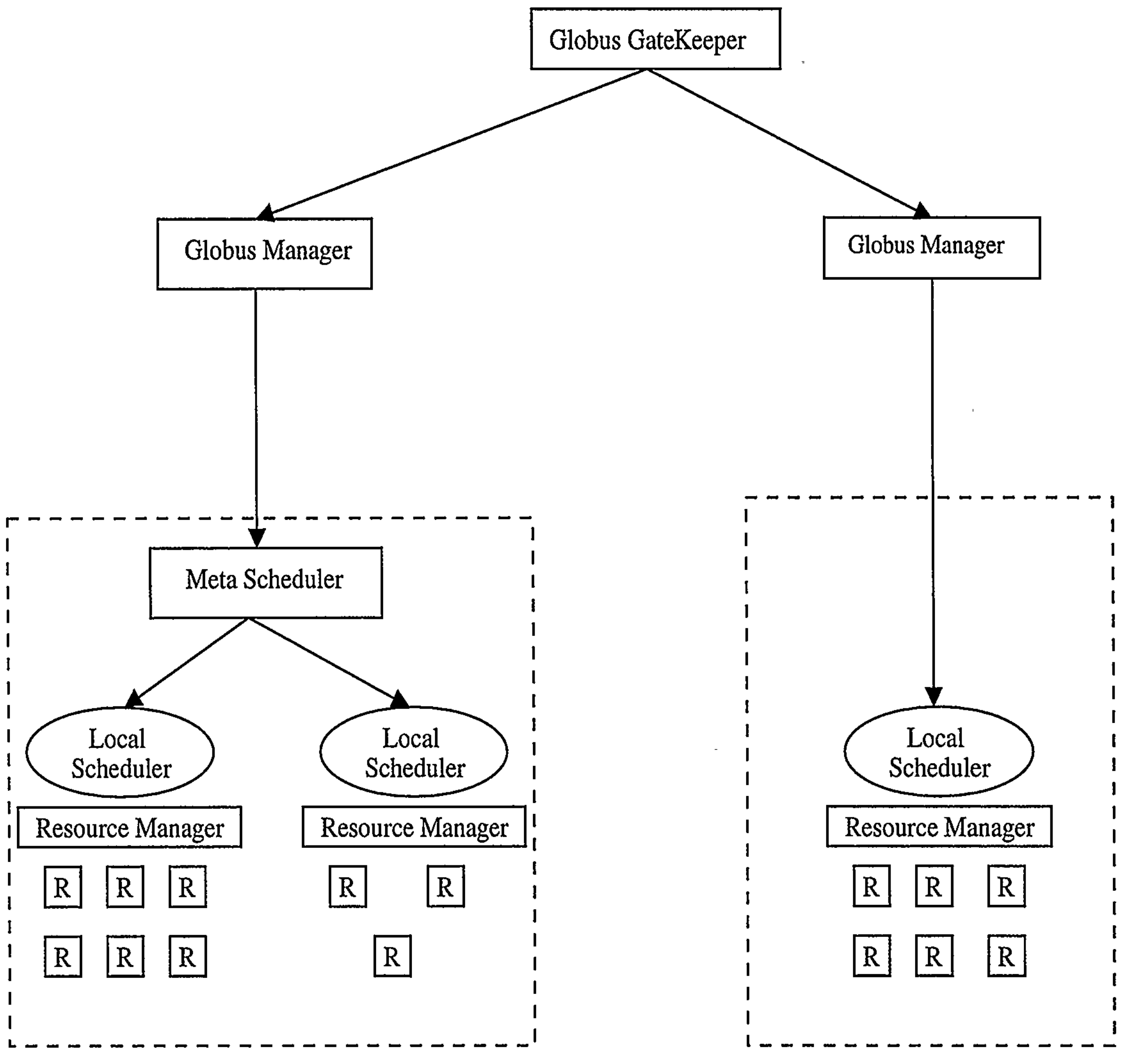
The difference with the Globus implementation is that, the job can be executed on any grid resource that meets the requirements of the job specified in the job script using RSL. Hence, the job is not limited to a particular computer or cluster of computers.

4.9 Summary

The *Wrapper method* for monitoring jobs in grid computing environment is described in detail in this chapter. The method would help users to understand how their jobs behaved on a particular computer by considering the monitoring information generated during the monitoring process.

The importance of monitoring information like CPU utilization, network activity, memory usage, disk space, disk I/O operations, shared file system activity, and process status statistics were discussed. The various sources of monitoring data were described; the advantages and disadvantages of getting monitoring data from those sources were also discussed.

The method by which the monitoring data is transferred to locations accessible



Note: The Local or Meta Scheduler could be PBS, Condor–G, LSF, LoadLeveler, NQE, etc.

Figure 4.3: Execution of Jobs in Globus-managed Grid Computing Environment

by users was described. In addition, the method of presenting the monitoring information to users was highlighted. This chapter further described how the *Wrapper method* is implemented in this thesis. The issues with how the *Wrapper method* is implemented in this thesis were also highlighted. Finally, how the *Wrapper method* would be used in a grid computing environment with Globus middleware was described.

Chapter 5

Experiments and Results

The purpose of the experiments described in this chapter is to confirm that the monitoring information shows the activity that takes place during the life cycle of a job on a computer. The monitoring information would help users in understanding the behaviour of their jobs when their jobs are sharing computing resources with other jobs in a computing environment. The experimental design, workload, and testbed used in this thesis are also described in this chapter.

Section 5.1 states the purpose of the experiments. Section 5.2 describes the methodology used in conducting the experiments. The description includes the experimental design, the workload design, and the environment under which various experiments were carried out. The experiments were carried out on a cluster of computers and the corresponding results are presented in Section 5.3. The importance of monitoring information is highlighted in each experiment. In addition, bottlenecks are identified and suggestions are made on how to improve the performance of jobs in different circumstances. The validation of the *Wrapper method* based monitoring tool is discussed in Section 5.4. Finally, the chapter is summarized in Section 5.5.

5.1 Purpose of Experiments

The experiments are designed to confirm the importance of the monitoring information generated using the *Wrapper method.* The emphasis of the experiments is on understanding what happened to a job on a particular computer and how the monitoring information could help in understanding why the job behaved in a certain way. In this thesis, eight experiments were carried out under different conditions and the monitoring information collected in those experiments was examined.

The experiments were designed in such a way that they show how the monitoring system and results would help users to understand the behaviour of their jobs on a computer. Attempts are made to answer the following questions in the experiments:

- Did a job get the expected amount of system resources?
- Is a job affected by other resource-consuming tasks or processes?
- What happened to a job during its life cycle on a computer?
- Why did a job behave in a certain way on a computer at a particular time?
- What are the bottlenecks to the performance of a job on a computer?
- How will the monitoring information help a user to improve the performance of his/her job?

5.2 Experimental Methodology

The design and implementation of *Wrapper method* for monitoring jobs in grid computing environments are described in Chapter 4. The experimental design, workload, and experimental testbed used in this thesis work are described in this section.

5.2.1 Experimental Design

The experiments conducted in this thesis work are designed in such a way that monitoring data is collected when a job is running on a computer. The monitoring data is transformed into meaningful monitoring information.

In these experiments, the Wrapper script is submitted to a batch scheduler, and it starts the job on a computer. The Wrapper script also starts the tool that monitors the job on the computer. The monitoring tool retrieves monitoring data about a job from the computer where the job is being executed. When the job is complete, the collected monitoring data is transformed into meaningful monitoring information by a graphing tool described in Chapter 4.

In some experiments, a computing task that uses a large amount of system resources is started on the same computer where the job is running in order to introduce competition for system resources. In order to understand the monitoring information generated using the *Wrapper method*, eight different experiments were designed in this thesis work.

Several runs of each experiment were carried out on the testbed. The results from each run was consistent with others in each experiment, so only the result of a single run is reported for each experiment.

5.2.2 Workload Description

Two computing tasks were used as workloads in the experiments conducted in this thesis. The tasks are computer programs running on a computer. The first program uses large amount of memory resources. The second program does file I/O using local disk, network, and shared file system resources. The first program is called memory workload and the second program is called file I/O workload.

5.2.2.1 Memory Workload

The pseudo code in Figure 5.1 describes the program in the first experimental workload. The purpose of the memory workload is to help in understanding how jobs use memory resources on a computer. In order to keep this workload simple, a large amount of memory is allocated to an array of integers. In the first phase of the program, random numbers are generated and stored in the array. Some elements of the array are read and modified in the second phase of the program. The storing of data in the array, reading and modifying elements in the array make sure the workload uses the memory resources on a computer.

The operating system allocates 2.4 Gigabytes of memory to an array of integers in the first phase of the memory workload. The total amount of physical memory on the host computer that will be used for running experiments in this thesis is 2 Gigabytes. 2.4 Gigabytes of memory is allocated to the array of integers so that the program would use more memory than the physical amount of memory on the computer. In this workload *malloc* function is used to allocate memory space to hold the array of integers. *malloc* is one of the library functions used by a program to manage the memory resources on a computer. The *malloc* library function calls *sbrk* when a process runs out of memory. *malloc* will fail when *sbrk* does not return memory to it.

If *malloc* succeeds in allocating 2.4 Gigabytes of virtual memory to the array, *Number of Elements* random numbers are generated and stored in the array. *Number of Elements* is the number of integers that would fit into 2.4 Gigabytes of memory and this value depends on the architecture of the processor - 32bits or 64bits. The architecture of the processor determines the amount of memory needed to process

```
Allocate 2.4 Gigabytes of Memory to Array of Integers by calling malloc() function

First Phase
FOR (counter = 1 TO Number of Elements) DO
{
  Generate a random number by calling rand() function
  Store random number in Array of Integer location [counter-1]
  Calculate the sum of generated random numbers
}

Second Phase
FOR (page = 1 TO Number of Memory Pages) DO
{
  Access an Element in the Memory Page
  Generate a random number by calling rand() function
  Update the Value of the Element by adding the generated random number to it
  Calculate the sum of Updated Elements
}

Print the sum of random numbers generated in Phase 1
Print the sum of Updated Elements computed in Phase 2
```

Figure 5.1: Description of Memory Workload

and store an integer on a computer.

Random numbers are generated by using *rand* function. The generated random numbers are stored in the array of integers, and the sum of the generated random numbers is computed. The sum of the generated random numbers is kept until the end of the program when it is printed out. The reason for keeping the sum of the random numbers is to prevent the random number generation code segment from being removed by the compiler during optimization.

In the second phase of the program, the *Number of Memory pages* is the number of memory pages that would fill 2.4 Gigabytes of memory. This value depends on the memory page size which in turn depends on the architecture of the processor on a computer. The following steps are taken for each memory page that belongs to the job. The program retrieves an element in the memory page; generates a random number using *rand()* function as in the first phase of the workload; updates the value of the retrieved element by adding the random number to it; computes the sum of the updated elements. The sum of the updated elements is printed out at the end of the program.

In this thesis, the memory workload is implemented on Linux-based computer; and 600,000,000 random numbers are generated and stored in the array of integers. The program was implemented on a Linux computer with a 64-bit AMD Opteron processor. The size of an integer on the 64-bit AMD Opteron processor is 4 bytes. In addition, the physical memory on the computer is divided into 4 Kilobytes of memory pages. Therefore, about 1,000 integer values can be stored in 1 page of memory with the memory header preceding the first page of the memory block.

5.2.2.2 File I/O Workload

The second experimental workload consists of a file Input/Output program. The program transfers the contents of an existing file in chunks to a new file. The data transfer is carried out using a buffer of fixed size k bytes. The data is transferred in chunks until all the content of the existing file is transferred into a new file on the same or different file systems. This workload program uses low-level file I/O functions including *open*, *read*, and *write*. The pseudo code in Figure 5.2 describes the workload used for the experiments that demonstrates job file system Input/Output activity.

```
{
    Create a buffer of k bytes
    Open an Existing File for reading
    Create and Open a New File for writing

    While not End of File
    {
        Read next k bytes of Existing File into buffer
        Transfer the contents from the buffer into the New File
    }
    Close New File
    Close Existing File
}
```

Figure 5.2: Description of File Input/Output Workload

In this thesis a file containing 8.4 Gigabytes of data is copied from a source to a destination. The source and destination could be local disk volume or remote disk volume through a shared file system. Data is transferred from the source to the destination using a 10-Megabyte buffer.

Each workload has a program known as the job and the other known as the competing task. The wrapper script starts the job; the user starts the competing task on the computer where the job is running. The job and the competing task will use the same computer program, so that neither the job nor the competing task would have an undue advantage over each other.

5.2.3 Experimental Testbed

All the experiments in this thesis work were carried out on a computer cluster. The computer cluster has 11 computers; each computer in the cluster runs the Linux operating system. The version 3 of Sun Microsystems' shared file system known as Network File System (NFS) is installed on the cluster. The NFS has a server and a client side, and the primary function of the NFS is to export or mount directories to other computers. NFS enables the file system volume that is physically residing on one of the computers to be visible and accessible from any other computer within the cluster.

Each computer in the cluster has 2.2 GHz AMD Opteron processor, 2 Gigabytes (GB) of physical memory, and 2 GB of swap space. The swap space is specially designated areas of disk used for paging. UNIX systems have a dedicated partition called the *swap partition* used for holding pages written out from memory.

PBS [85] resource manager is used as the batch system and MAUI [17] is used as the batch system scheduler in this environment. The Wrapper script is submitted to PBS, and MAUI decides where to execute the job on the cluster, except the user specifies a computer to use.

5.3 Experimental Results

Eight experiments were carried out under various circumstances, in order to see if the monitoring information collected about jobs would help in understanding the behaviour of jobs on a computer.

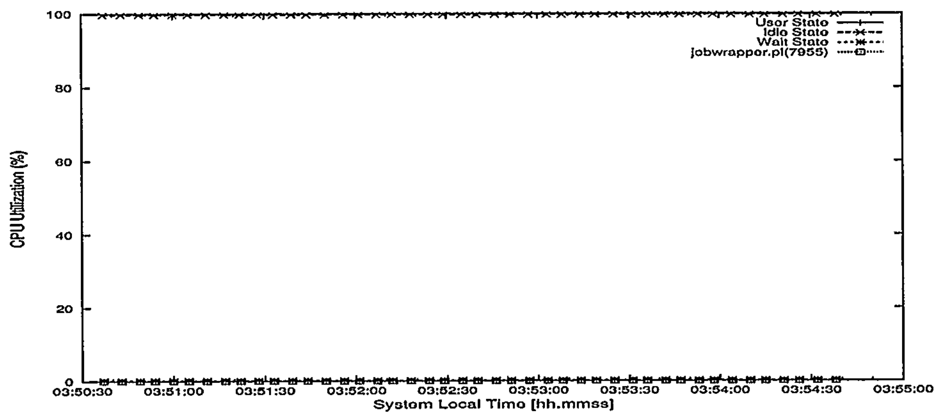
5.3.1 Experiment 1: Baseline Experiment

The aim of this experiment is to see if the monitoring information would show the state of the resources on a computer when it is not running any job. The results from this experiment will be used as a baseline when investigating the behaviour of jobs on the same computer.

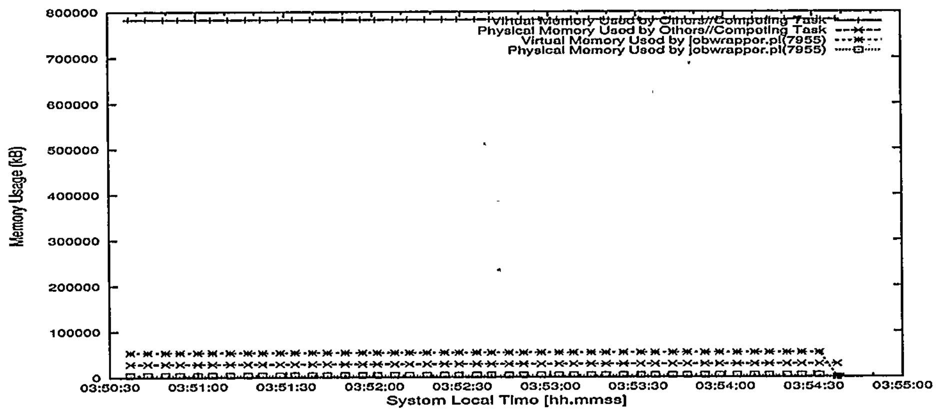
In this experiment, the job was designed to *sleep* (i.e., do nothing) for 240 seconds. The job did not use system resources during this period of time. The result in Figures 5.3 and 5.4 show the state of the computer used in this thesis when a user's job is not using system resources.

Figure 5.3(a) shows that the job did not use any CPU time. Hence, the job's CPU utilization is 0% and the CPU idle time is about 100%. It is shown in Figure 5.3(b) that the job used very little memory on the computer. Figure 5.3(c) shows that 1 minor page fault occurred per second, and no major page fault occurred. This suggests that very little memory was used by the job, or else there would have been a significant number of minor page faults.

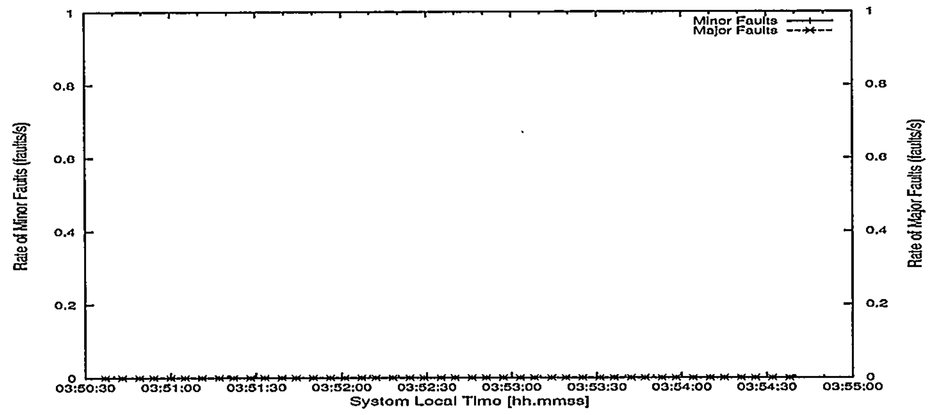
Figure 5.4(a) shows that an average of 300 kB of data was written from the physical memory to the local disk volume per second, but no data was read from the local disk into the physical memory. The disk I/O information shows all the local disk I/O activity on the host computer. The cause of disk I/O activity is not known in this experiment. The network activity result in Figure 5.4(b) shows that 0.1 byte of data is written and read from the network interface per second. This is the overall network activity on network interface *eth0* of the computer. About 24 bytes of data was written by the shared file system client per second, and no data was read by the shared file system client as shown in Figure 5.4(c).



(a) CPU Utilization

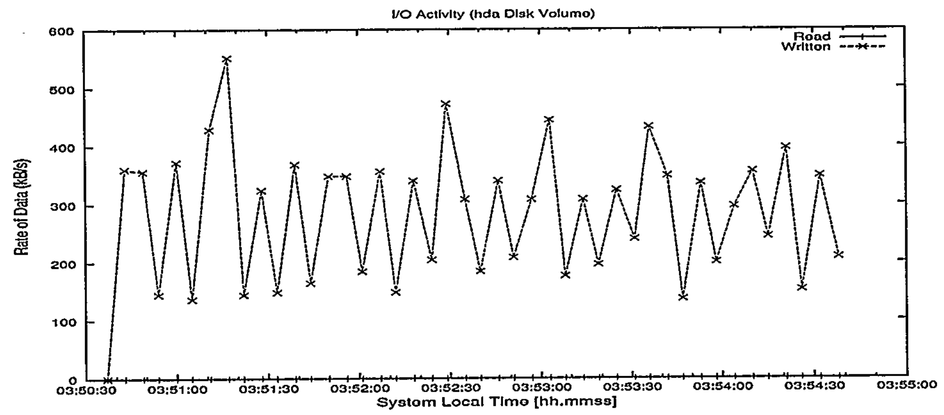


(b) Memory Usage

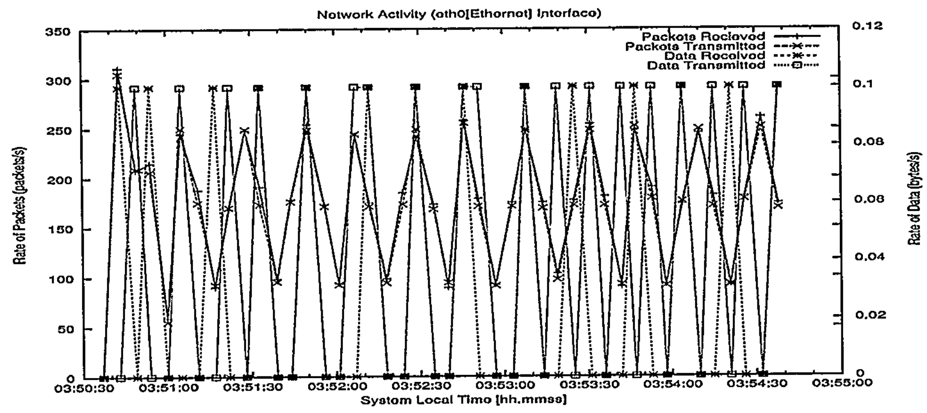


(c) Page Faulting

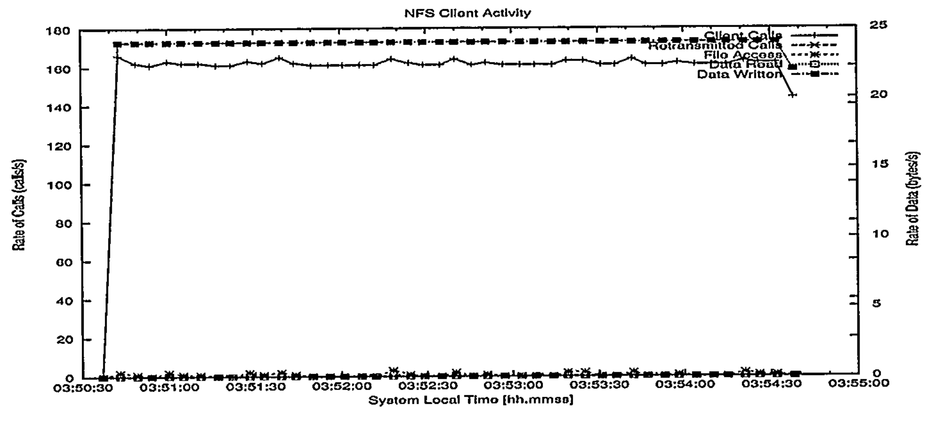
Figure 5.3: Job not using System Resources



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.4: Job not using System Resources

A network analysis is carried out in order to understand the disk I/O, network, and shared file system information in this baseline experiment. The *tcpdump* utility is used for monitoring the computer network traffic during another run of the baseline experiment. Eight hundred packets were collected and analyzed; the result indicating the number of packets transferred from a particular source to a unique destination is shown in Figure 5.5.

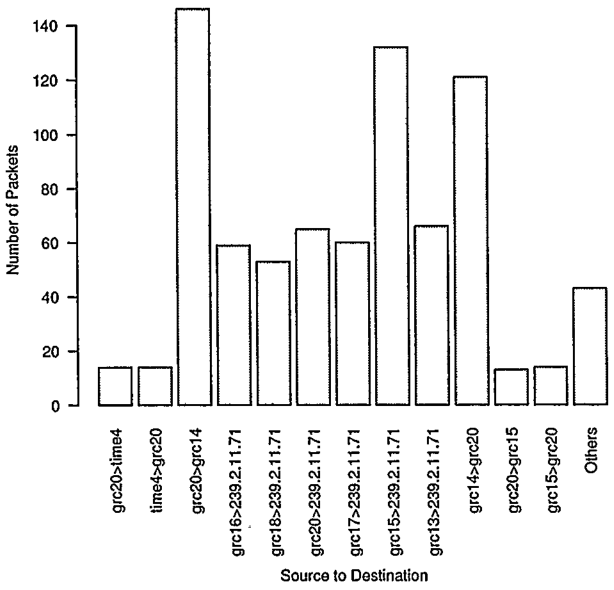


Figure 5.5: Source to Destination Network Analysis

The network analysis result shows that more than 140 packets were transferred from *grc20* to *grc14* and about 120 packets vice-versa. *grc20* is the host computer where this baseline experiment was executed, and grc14 is the computer running the NFS server. A significant number of packets were also transferred from some computers on the grc cluster to IP address 239.2.11.71. The Ganglia monitoring daemon (also known as *gmond*) provides monitoring on a single cluster by sending and receiving data on the multicast channel 239.2.11.71. This information suggests that the network activity seen in the baseline experiment is caused by other applications not related to the job.

Since the purpose of this experiment is to show the state of the host computer when it is not running any job, therefore the result in Figures 5.3 and 5.4 would serve as a reference point when describing the experimental results in this thesis.

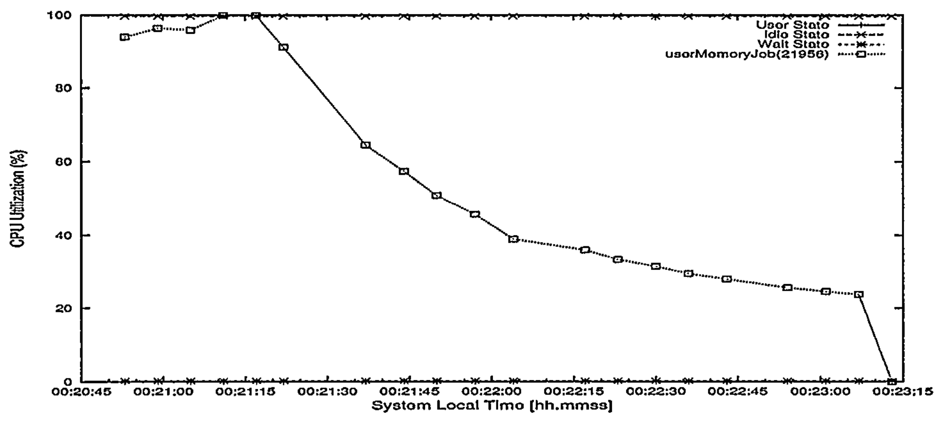
5.3.2 Experiment 2: Memory-Intensive Job not Competing for System Resources

The aim of this experiment is to see if the monitoring information gathered about a job that uses memory resources on a computer could show that the job is not experiencing competition from other tasks on the same computer.

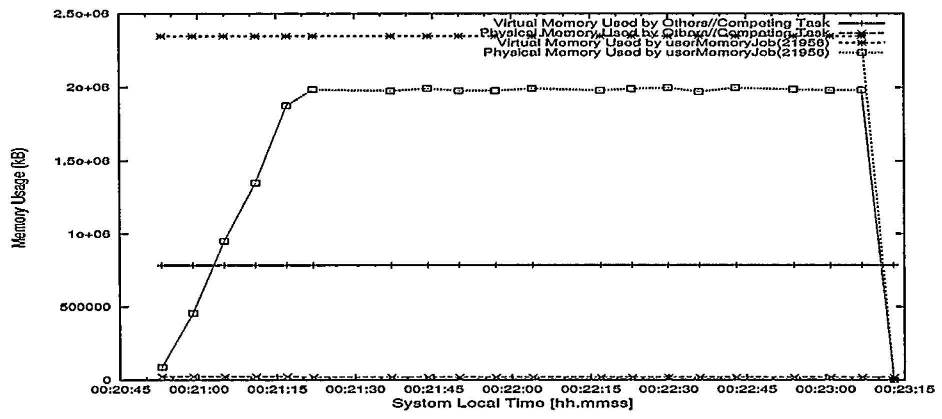
In this experiment, the job was executed on a computer with about 2 Gigabytes of unused physical memory space. The memory workload which demonstrates the use of memory resources on a computer is used in this experiment.

Figures 5.6 and 5.7 show the monitoring information of the job on the computer where it was executed. The monitoring information in Figure 5.6(a) shows that the job received about 95-100% CPU processing time between 00:20:45 and 00:21:20. After 00:21:20, the CPU utilization decreased suddenly to about 65% at 00:21:35. Then, the CPU utilization decreased gradually to about 25% towards the completion time of the job. Figure 5.6(b) shows that the physical memory used by the job increased until 00:21:20, then it remained constant until the job was completed. The figure also shows that the virtual memory used by the job was constant throughout the life cycle of the job.

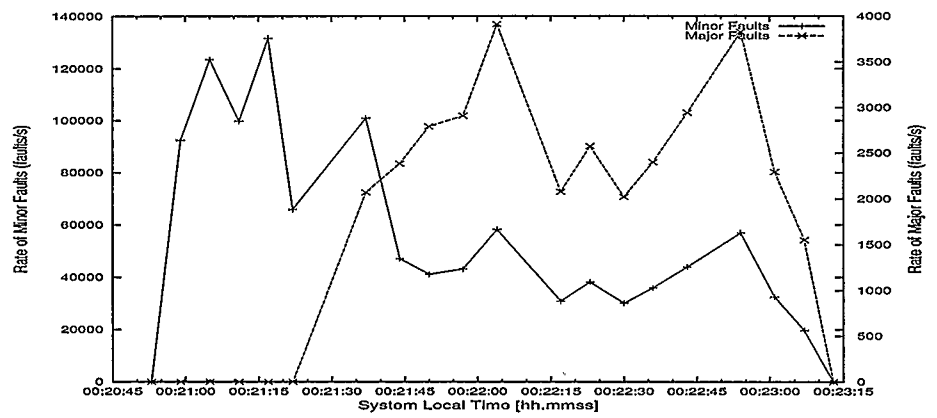
A huge number of minor and major page faults occurred during the execution



(a) CPU Utilization

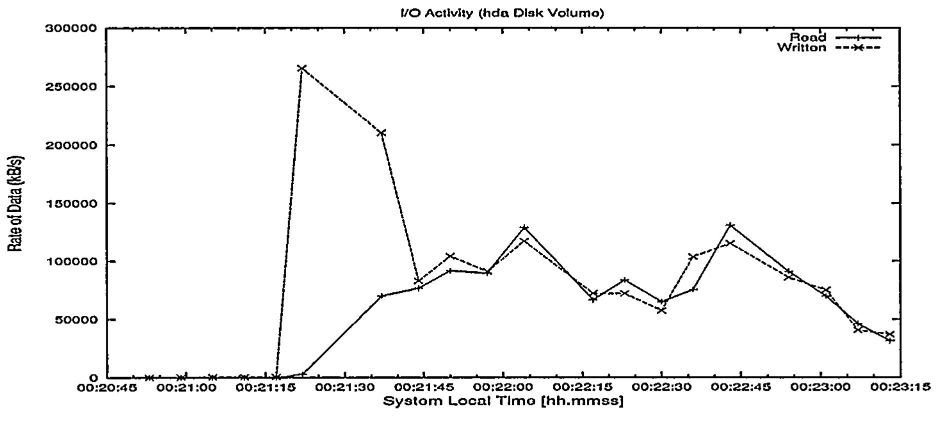


(b) Memory Usage

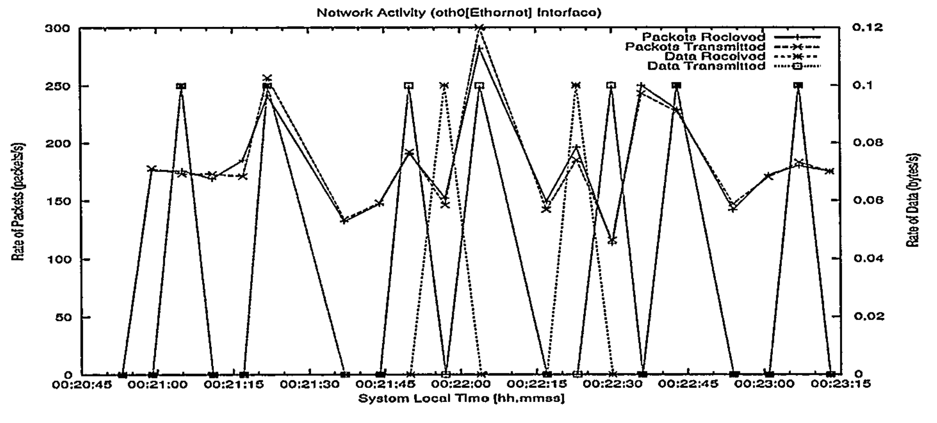


(c) Page Faulting

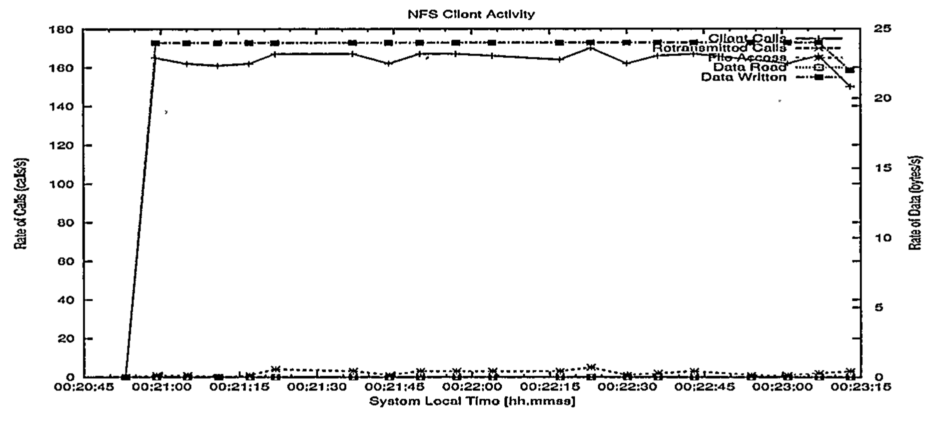
Figure 5.6: Memory-Intensive Job not Competing for System Resources



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.7: Memory-Intensive Job not Competing for System Resources

of the job as shown in Figure 5.6(c). Minor page faults occurred at a higher rate from 00:20:45 to 00:21:35 because the virtual memory pages are not mapped in the physical memory. No major page fault occurred during this time since no page is being copied from disk to the physical memory. The rate at which minor page faults occurred decreased after 00:21:35 until the job was completed. This marks the end of the first phase of the workload where random numbers are stored in an array of integers. The second phase of the workload starts at about 00:21:35 with major page faults.

Major page faults started at 00:21:35; it continued until the job was completed. From 00:21:35 onwards, memory pages are being read and written to in the second phase of the program starting with the first memory page. All the memory pages that belong to the job can not be stored in the physical memory at the same time. Therefore, stale memory pages are copied from the physical memory to disk, while the memory pages needed by the job are brought into the physical memory. Therefore, when the program makes an attempt to access the first page, the first page would not be found in the physical memory, so it would be fetched from disk. This results in a major page fault since the local disk is accessed when the fault occurred.

Figure 5.7(a) indicates that the local disk was not accessed at the beginning of the first phase of the program i.e., from 00:20:45 to 00:21:20. The operating system started writing memory pages to disk and reading memory pages from disk between 00:21:20 and 00:21:25. At this point, the physical memory is exhausted and memory pages are being written to disk in order to create space for new memory pages. Between 00:21:25 and 00:21:45 less data is being written but more data is being read from disk per second. From 00:21:45 until the job was completed, about the same amount of data is being read and written to disk. This behaviour depicts the second phase of the program where memory pages are being read and updated. In the second phase, an old memory page is copied to disk in order to bring in the needed memory page from disk into the physical memory.

Data was transmitted across the computer network through network interface *eth0*. Figure 5.7(b) shows that the same amount of data is being transmitted and received through the network interface. Similarly, about the same number of packets is transmitted and received over the network interface. This behaviour is similar to that observed in Figure 5.4(b) in the Baseline experiment. Therefore, this result indicates that the job did not use network resources.

The result in Figure 5.7(c) shows there were a number of calls from the NFS client on the host computer to the NFS server; about 24 bytes of data was written by the NFS client per second. The amount of data written was constant, and the number of client calls did not change much throughout the life cycle of the job.

The overall results show that the job received about 95% CPU time before the physical memory was exhausted. The CPU utilization started to decrease after the physical memory was exhausted at 00:21:20. The CPU utilization of the job decreased because CPU time is being used to move memory pages to disk and vice-versa. Hence, the job was spending more time in the *blocked state* while the memory pages are being written to local disk volume.

The results in Figures 5.6 and 5.7 show that the job did not experience competition for CPU, memory, and file system resources from other processes on the computer. In addition, the monitoring information shows that the bottleneck for the job is the physical memory. The computer has a physical memory of 2 Gigabytes and the job used about 1.9 Gigabytes of the physical memory. The decrease in CPU processing time and the occurrence of major page faults when the physical memory was exhausted confirms physical memory as the bottleneck.

Using this monitoring information, the user can submit his/her job to another computer with more physical memory next time. The monitoring information obtained for this experiment shows that there was little contention for computing resources between the job and other tasks on the computer, but the physical memory was a bottleneck.

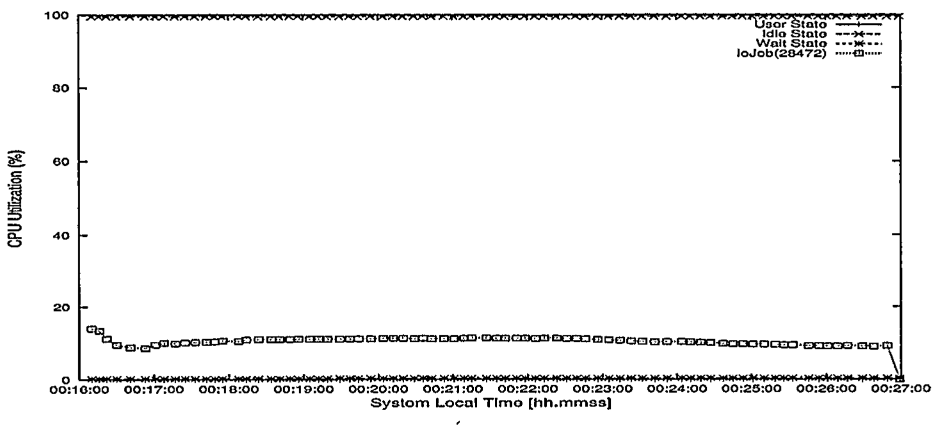
5.3.3 Experiment 3: File I/O-Intensive Job (between Remote Volumes)

The objective of this experiment is to see if the monitoring information would reveal that a job is performing file I/O operations on a remote volume through a shared file system on a cluster of computers.

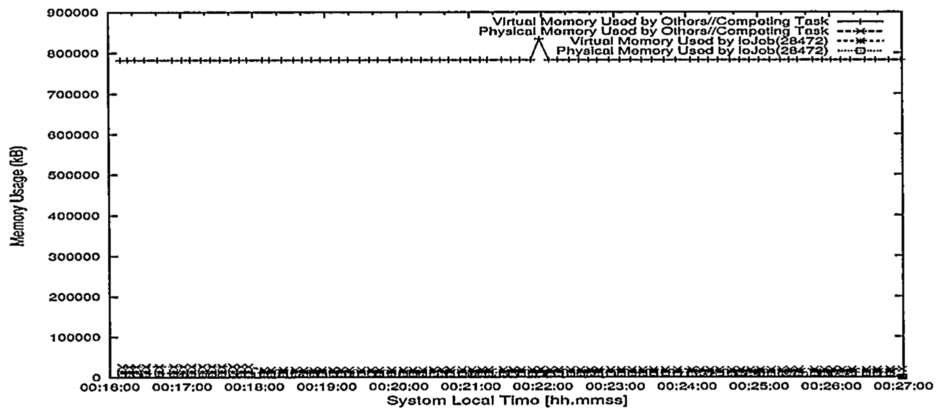
The workload used in this experiment is designed to make a copy of an existing file into a new one on a shared file-system volume. The workload is described in detail in Section 5.2.2. The job is submitted to a cluster of computers and the monitoring information is considered in order to understand how the job behaved.

Figures 5.8 and 5.9 show the monitoring information of the job on the computer where it was executed. The monitoring information in Figure 5.8(a) shows that the job used about 10% CPU processing time when it was running on the computer. The job used very little memory as shown in Figure 5.8(b). The page fault information in Figure 5.8(c) shows a similar behaviour to the one observed in the baseline experiment.

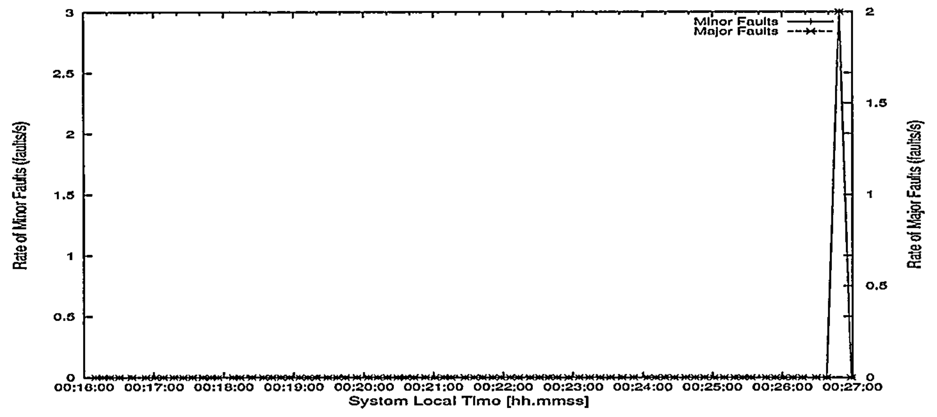
The disk I/O result in Figure 5.9(a) shows that a small amount of data was read and written to disk throughout the lifetime of the job. Figure 5.9(b) shows that



(a) CPU Utilization

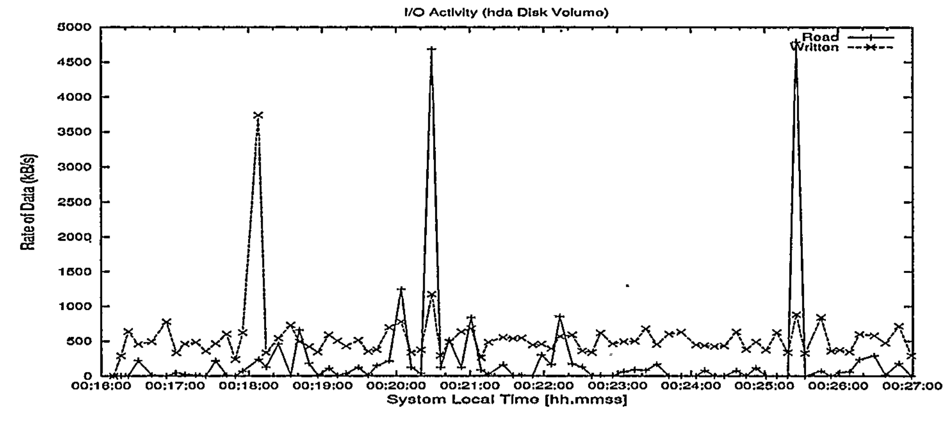


(b) Memory Usage

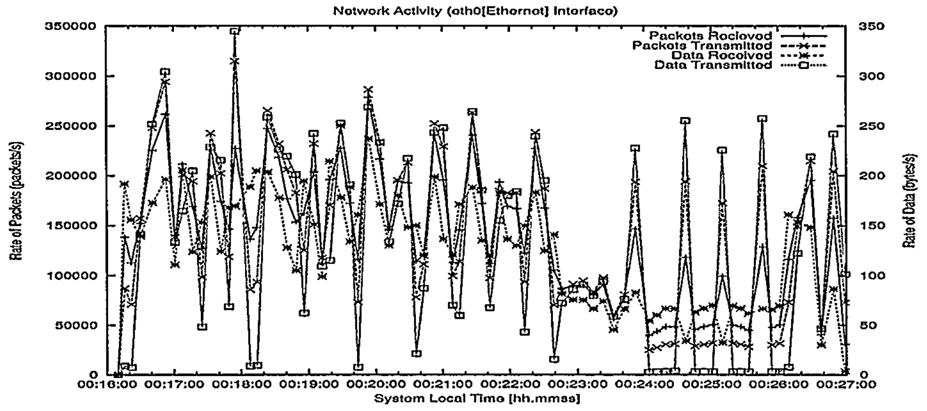


(c) Page Faulting

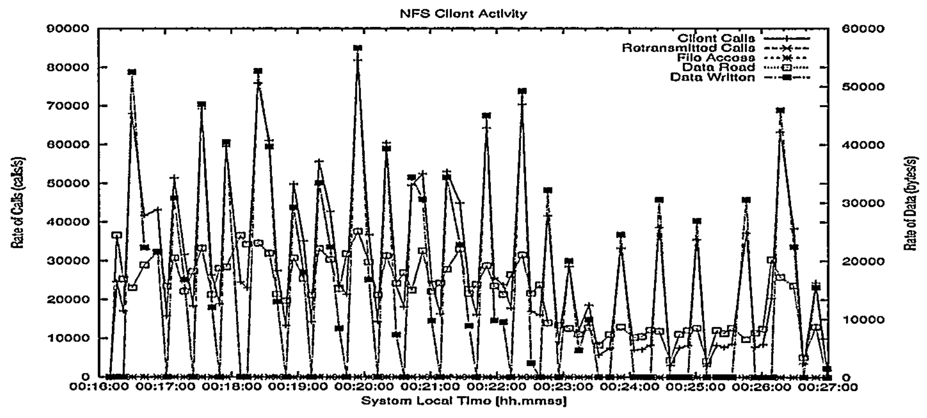
Figure 5.8: File I/O Job between Remote Volumes



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.9: File I/O Job between Remote Volumes

the network interface *eth0* was busy sending and receiving packets. Figure 5.9(c) shows the information from the client side of the shared file system; it shows that a huge amount of data was read and written to the shared file system volume. The information in Figures 5.9(a), 5.9(b), and 5.9(c) show that data is being read and written to the shared file system volume as the experiment progresses. Other activities on the computer not related to the user's job is likely to have caused the spikes in the disk I/O activity

The combined monitoring information gathered in this experiment indicates that the job is likely to have caused the file I/O. It also shows that the file I/O occurred on a remote volume through the network. The monitoring information from this experiment shows that the shared file system could be a bottleneck in this experiment. Hence, if the job is not completed at the expected time, the user can submit the job to another computer with faster shared file system some other time. A busy shared file system may cause the job not to be completed on time. In this case, the user may choose to submit his/her job to the same computer at a later time when the shared file system is less busy.

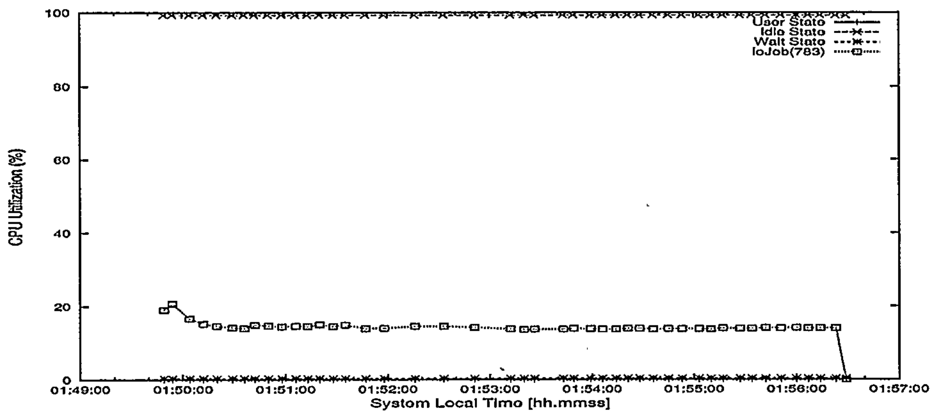
5.3.4 Experiment 4: A File I/O Job Competing for Resources with Memory-Intensive Task

The goal of this experiment is to determine if the monitoring information would show the user that his/her job is competing for resources with other tasks on the same computer. Also, to see if the monitoring information could show what resources are being competed for, when the competition happened, and the effect of the competition on the user's job.

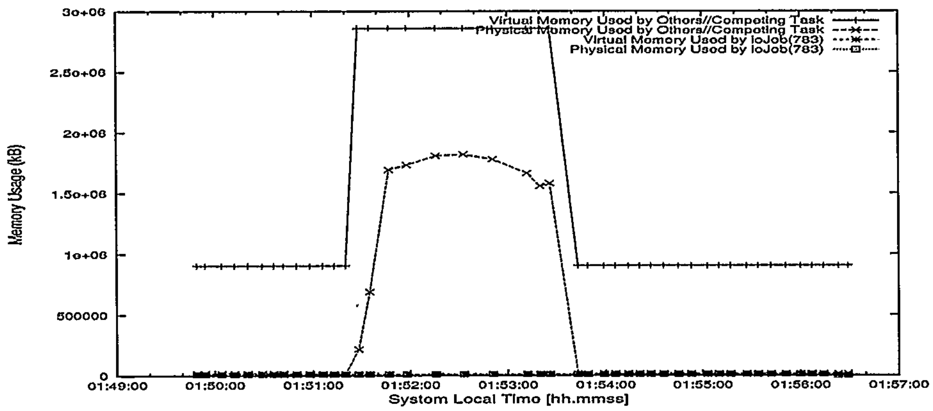
In this experiment the user's job is the file I/O workload and the competing task is the memory workload described in Section 5.2.2. The file I/O activity took place on a remote volume through the NFS. The job was executed on one of the computers in the *grc* cluster and allowed to run for about 2 minutes before the competing task was started on the same computer. Figures 5.10 and 5.11 show the results obtained in this experiment.

The CPU information in Figure 5.10(a) is similar to the one obtained for Experiment 3. The information indicates that the job is not CPU-intensive. Figure 5.10(b) shows that the physical memory and the virtual memory used by the job were constant throughout the life cycle of the job. But the virtual and physical memory used by other tasks on the computer increased between 01:51:00 and 01:54:00. The monitoring information shows that there were some tasks (external to the user's job) that caused the increase in virtual and physical memory. At that point there is competition for memory on the computer where the job was executed. The page fault information in Figure 5.10(c) shows that minor and major page faults occurred at about the time when the job started and finished. This information is related to the user's job and not the competing task(s). The disk volume I/O activity information in Figure 5.11(a), the network activity information in Figure 5.11(b), and the shared file system information in Figure 5.11(c) shows that there was increase in activity on the computer between 01:51:00 and 01:54:00.

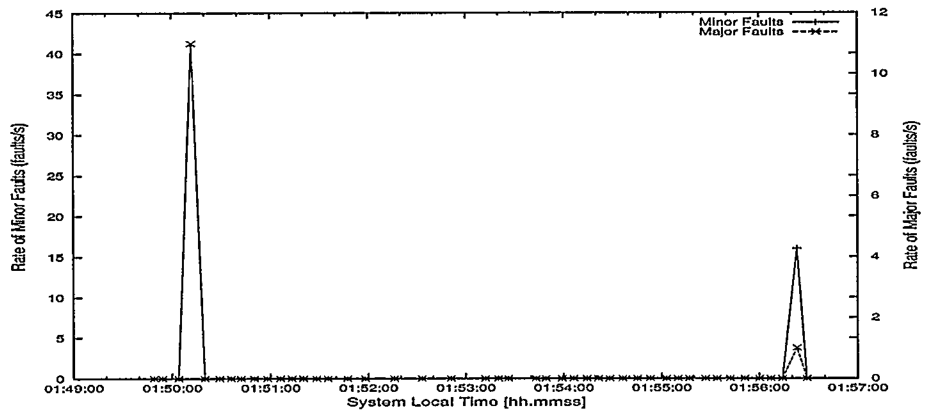
The monitoring information generated from this experiment shows that the job experienced competition for system resources including memory, disk I/O, network bandwidth, and shared file system volume. The monitoring information also showed when the competition occurred on the host computer. Using this information, the user may choose to submit his/her job to this computer some other time depending



(a) CPU Utilization

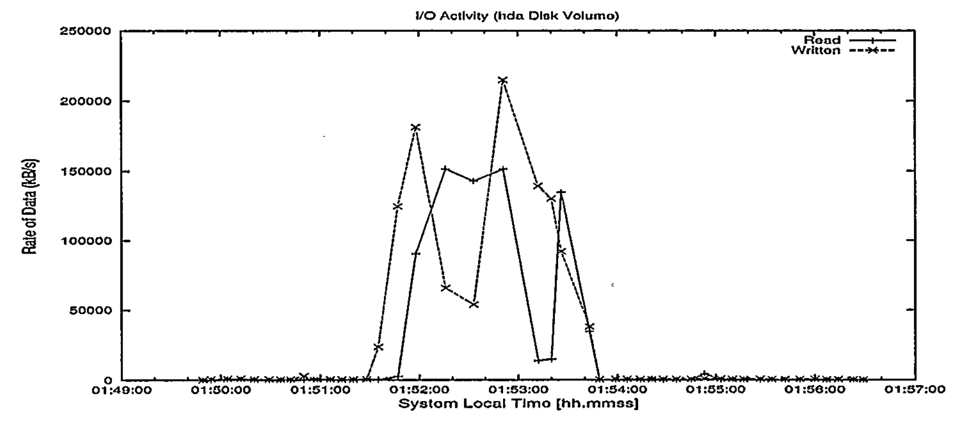


(b) Memory Usage

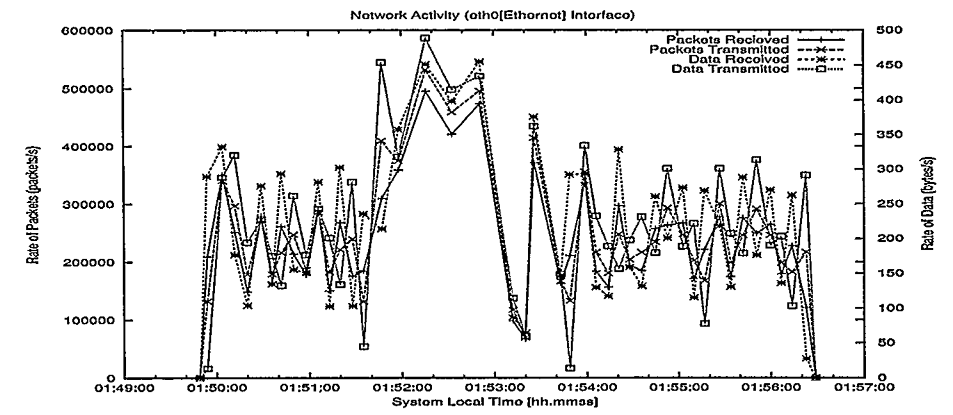


(c) Page Faulting

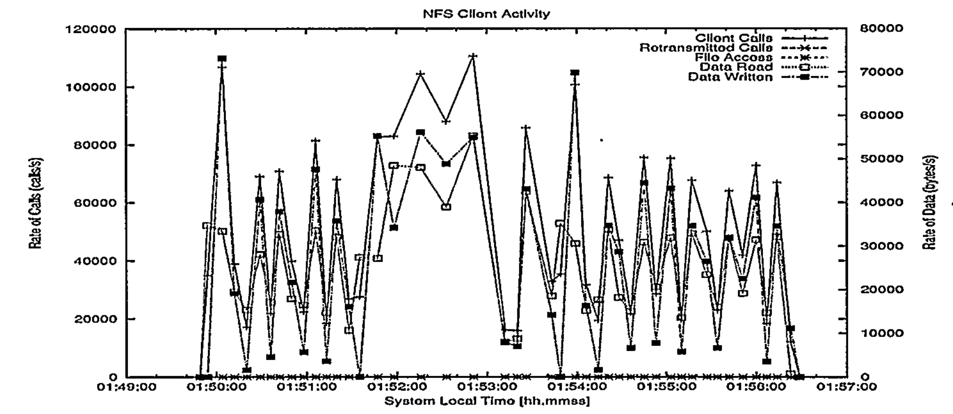
Figure 5.10: A File I/O Job Competing with Memory-Intensive Task



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.11: A File I/O Job Competing with Memory-Intensive Task

on the effect of the competition on the user's job.

5.3.5 Experiment 5: File I/O-Intensive Job (between Local Volumes)

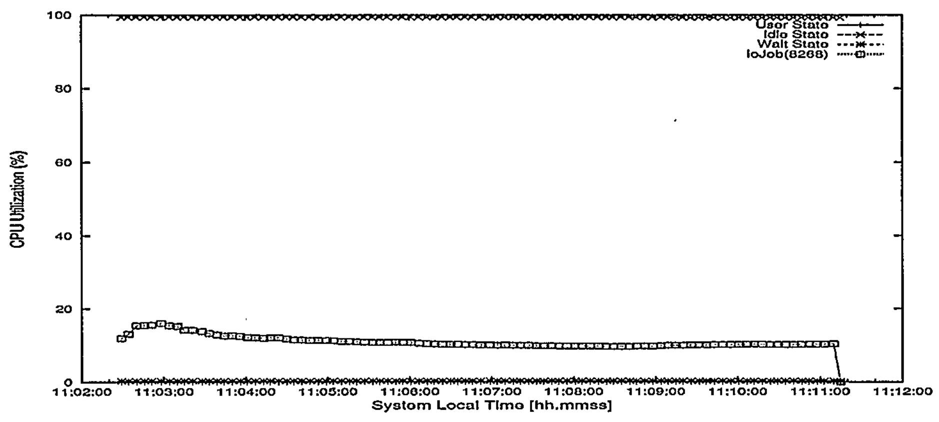
The aim of this experiment is to see if the monitoring information would reveal that the file I/O operations performed by a job are happening on a local disk volume, and not through a shared file system.

The workload used in this experiment is the same as the one used in Experiment 3. The only difference between this experiment and Experiment 3 is that the file I/O activity took place on the local disk volume in this experiment.

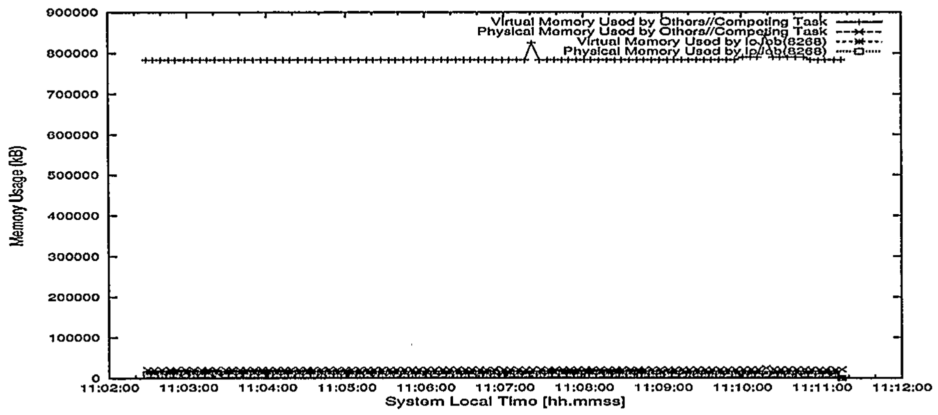
Figures 5.12 and 5.13 show the monitoring information collected about the job using the *Wrapper method.* The CPU utilization, memory usage, and page fault information in Figures 5.12(a), 5.12(b), and 5.12(c) respectively are similar to the ones in Experiment 3.

The disk I/O result in Figure 5.13(a) shows that about the same amount of data was read from disk and written to the local disk volume per second in this experiment. The pattern of file I/O activity observed in this experiment indicates that the file I/O activity happened on the local disk volume. That is, a certain amount of data is read from disk during a *read* operation, and the same amount of data is written to disk during the next *write* operation.

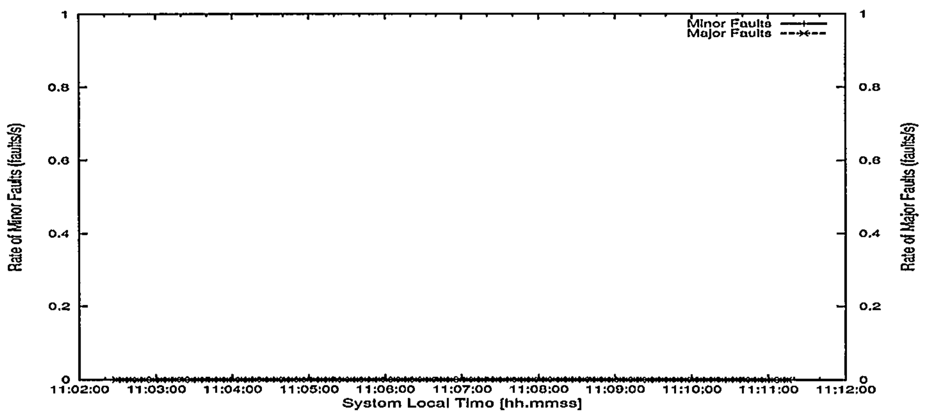
Figure 5.13(b) and Figure 5.4(b) (in the baseline experiment) look similar on the same scale; this indicates that the job used very little network resources on the host computer. Figure 5.13(c) is also similar to Figure 5.4(c) (in the baseline experiment); this is an indication that the job did not use the shared file system. Very little data is written to the shared file system compared to the result in Figure 5.9(c). This suggests that the little amount of data written to the shared file system may be



(a) CPU Utilization

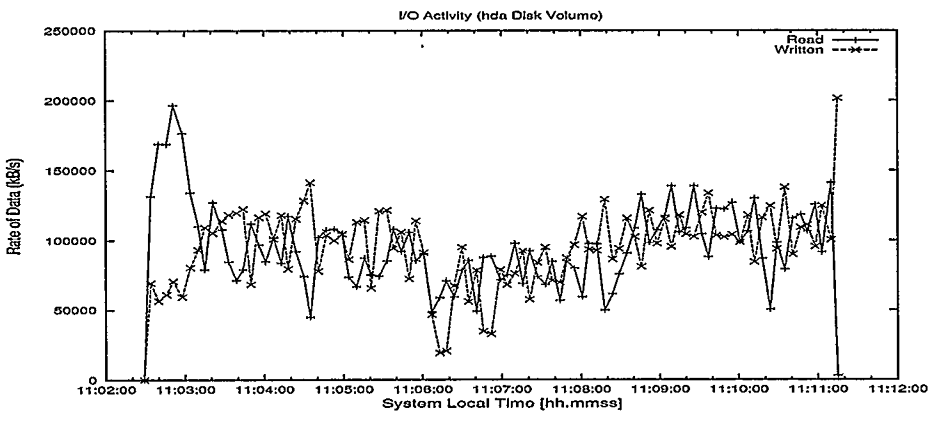


(b) Memory Usage

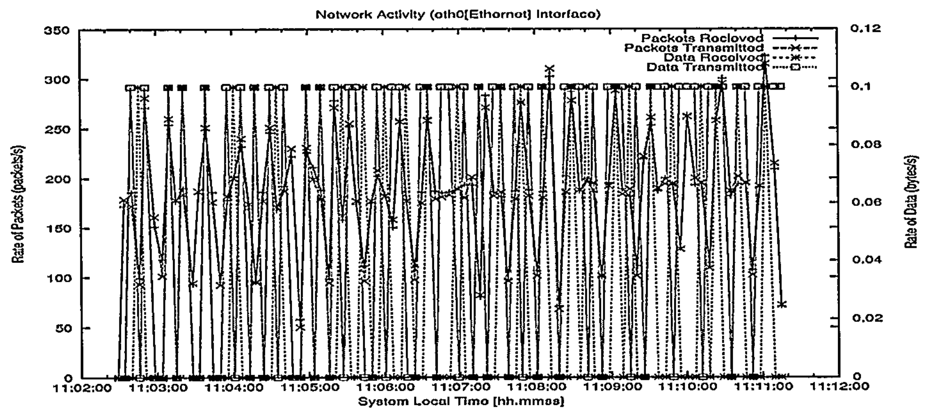


(c) Page Faulting

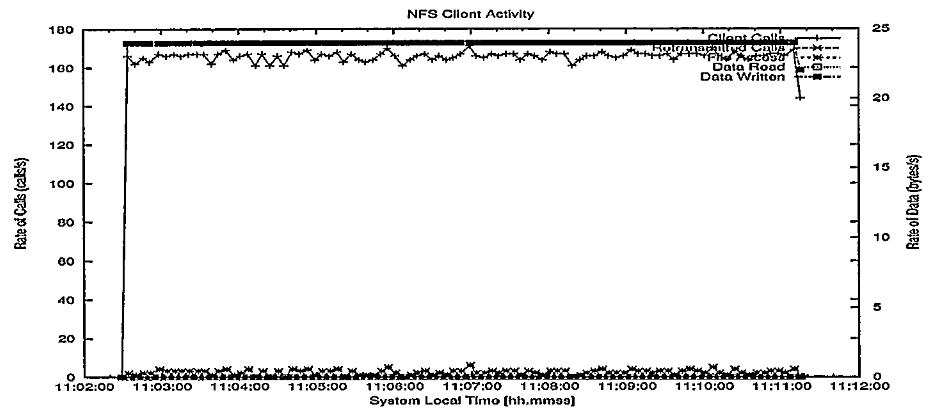
Figure 5.12: File I/O Job between Local Volumes



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.13: File I/O Job between Local Volumes

caused by other tasks on the host computer.

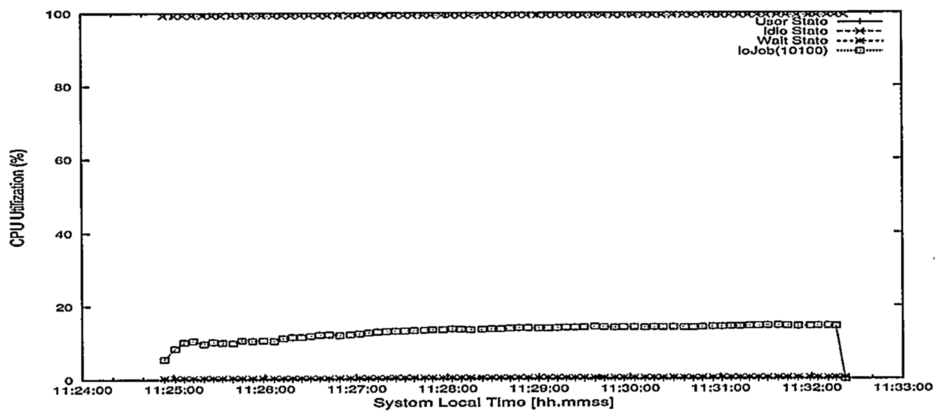
The information in Figures 5.13(a), 5.13(b), and 5.13(c) suggests that the job is likely to be responsible for the file I/O activity on the computer. It also shows that the file I/O happened on a local disk volume. In addition, the information in Figures 5.12(a), 5.12(b), and 5.12(c) indicate that the job in this experiment is likely to be a file I/O job. The combined monitoring information highlights the bottleneck of the job on the computer to be the local disk I/O. This job used the same workload as in Experiment 3, but the file I/O happened on a remote volume. With this information, the user may modify his job to use a remote volume next time because disk I/O is faster on the remote volume than the local disk volume on this cluster of computers.

5.3.6 Experiment 6: File I/O-Intensive Job (from Remote Volume to Local Volume)

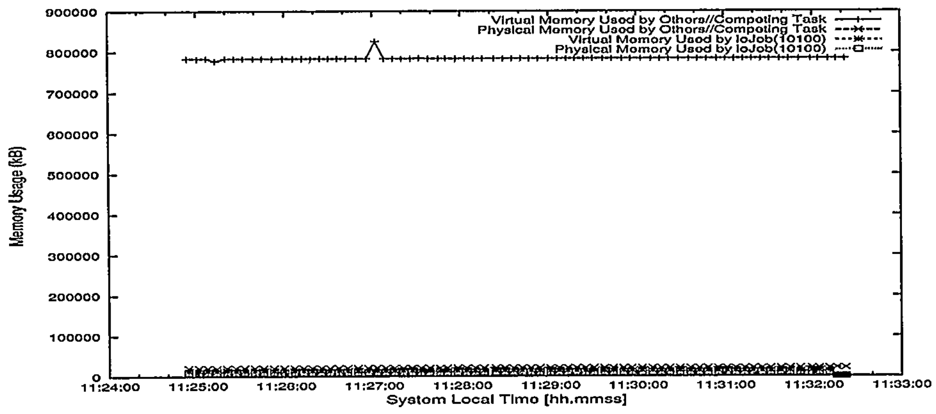
The aim of this experiment is to see if the monitoring information would show the user that the file I/O activity performed by a job occurred between a remote disk volume and a local disk volume.

The workload used in this experiment is the same as the one used in Experiments 3 and 5. The difference between this experiment, Experiment 3, and Experiment 8 is that the file I/O activity took place between NFS and a local disk volume in this experiment.

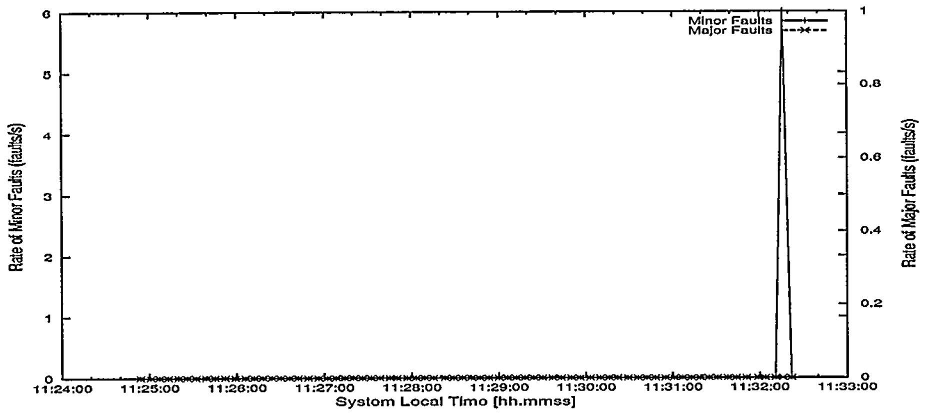
The monitoring information collected in this experiment is shown in Figures 5.14 and 5.15. The CPU utilization, memory usage, and page fault information in Figures 5.14(a), 5.14(b), and 5.14(c) respectively are similar to the results in Experiments 3 and 5.



(a) CPU Utilization

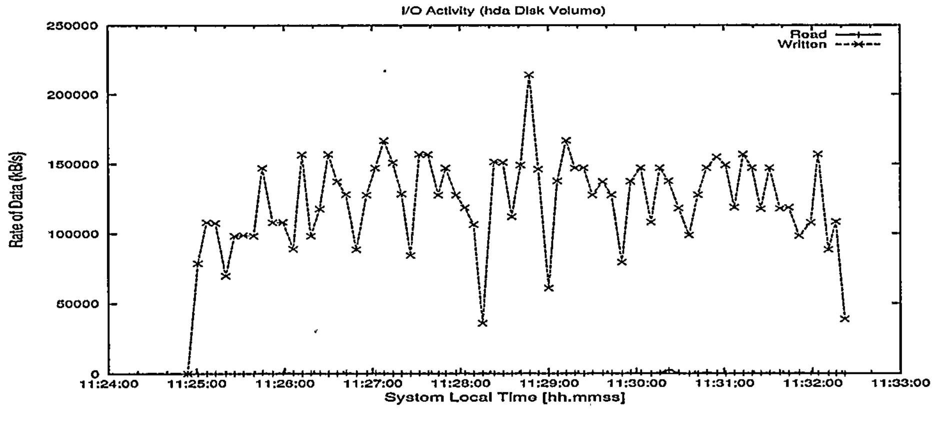


(b) Memory Usage

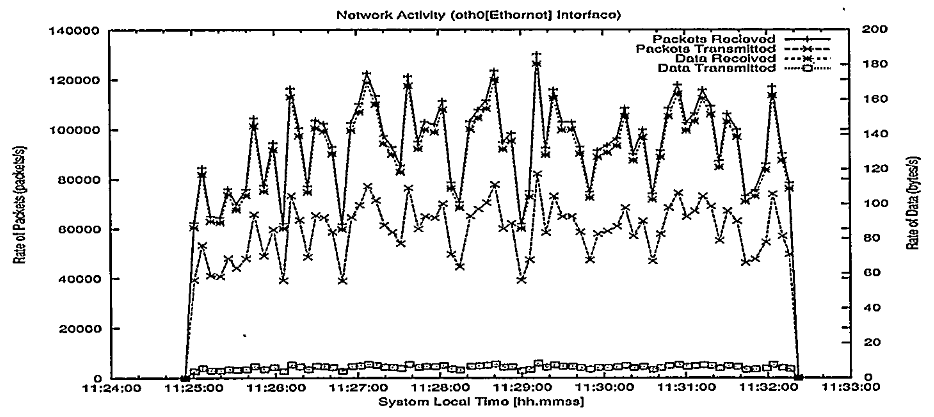


(c) Page Faulting

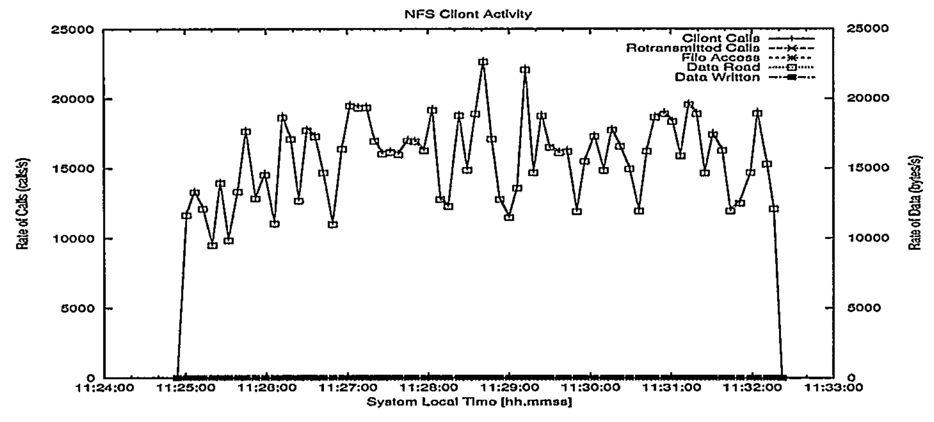
Figure 5.14: File I/O Job (Remote Volume to Local Volume)



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.15: File I/O Job (Remote Volume to Local Volume)

The disk I/O activity in Figure 5.15(a) shows that an average of 125 MBytes of data was written to the local disk volume per second, but very little data is read from the local disk volume. Figure 5.15(b) shows that the network interface *eth0* on the host computer was busy throughout the experiment. About 140 bytes of data is received on the network interface, but very little data is sent through the same interface. The shared file system activity in Figure 5.15(c) shows that data was read through the file system, but no data was written through it. The information from Figures 5.15(a), 5.15(b), and 5.15(c) confirms that both the local disk volume and the shared file system are involved in the file I/O activity.

The monitoring information considered in this experiment shows that the job is likely to be the cause of the file I/O activity. It also shows that the file I/O activity occurred between a shared file system volume and a local disk volume. The information further shows that the file I/O activity involves reading data from a shared file system volume and writing data to a local disk volume.

The monitoring information indicates the NFS and the local disk as bottlenecks for this job. If the user is not satisfied with the performance of the job, the user may modify his/her job to use a shared file system for the file I/O activity.

5.3.7 Experiment 7: File I/O-Intensive Job (from Local Volume to Remote Volume)

The purpose of this experiment is to determine whether the monitoring information would detect file I/O activity between NFS and a local disk volume. Also, to see if the monitoring information would show the source and the destination of the data.

The workload used in this experiment is the same as the one used in Experiment 6. The only difference is that data is read from a local disk volume and written to a shared file system volume in this experiment.

The monitoring information collected in this experiment is shown in Figures 5.16 and 5.17. The CPU utilization, memory usage, and page fault information in Figures 5.16(a), 5.16(b), and 5.16(c) respectively are similar to the results observed in Experiments 3, 5, and 6.

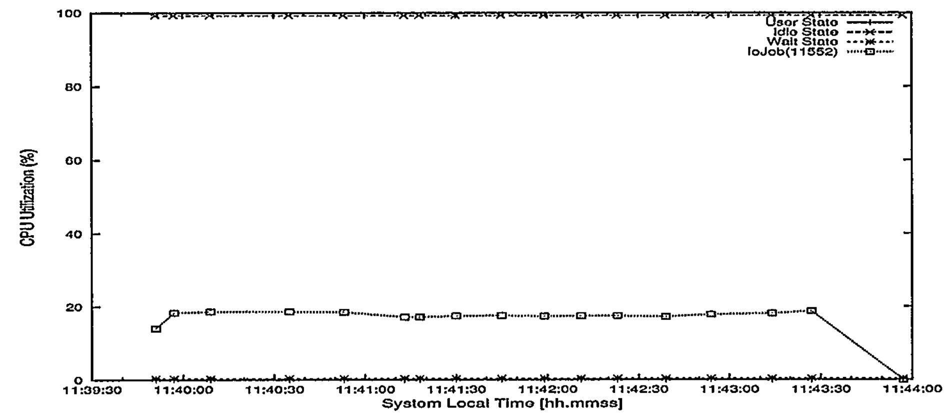
The disk I/O result in Figure 5.17(a) shows that data is read from the local disk, but very little data is written to the local disk. This result is the opposite of that observed in Figure 5.15(a) in Experiment 6. This result suggests that the local disk volume is used in this experiment. Figure 5.17(b) shows that the network was busy throughout the experiment; this is an indication that the shared file system may be involved in this file I/O activity like in Experiment 6. The result in Figure 5.17(c) shows that a significant amount of data is written to the shared file system but very little data is read from it.

The plots in Figures 5.17(a), 5.17(b), and 5.17(c) are similar. The combined results indicate that data is read from the local disk volume and written to the shared file system. The monitoring information shows that the job is likely to be responsible for the file I/O activity, which involves reading data from a local disk volume and writing the data to a remote disk volume through the shared file system.

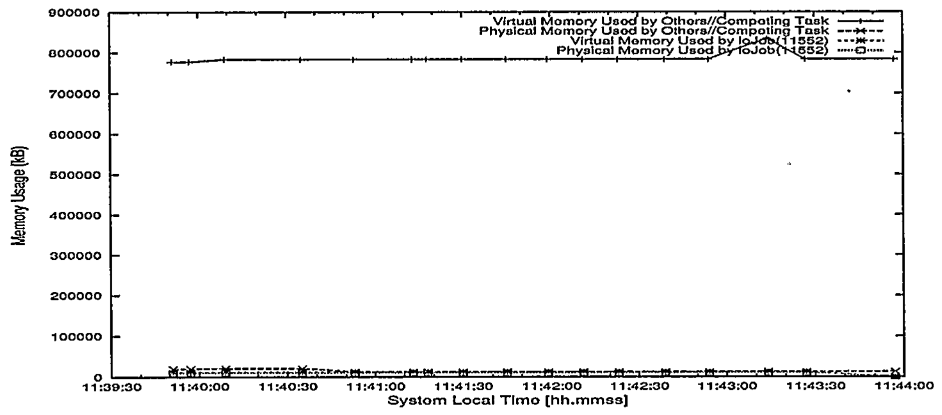
This result shows that the bottlenecks for the job are the shared file system and the local disk. In order to improve the performance of the job, the user may modify his job do the file I/O on a remote volume through a shared file system.

5.3.8 Experiment 8: A Job Competing with a Memory-Intensive Task

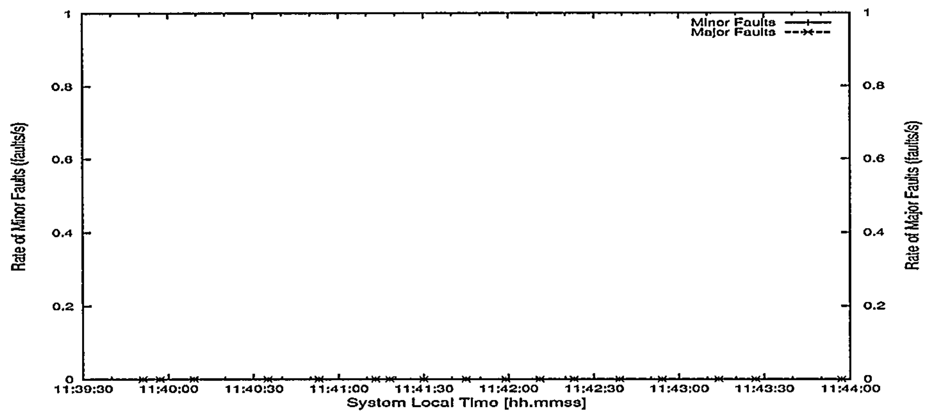
The goal of this experiment is to determine if the monitoring information can show a user that his/her job experienced competition for computing resources with another



(a) CPU Utilization

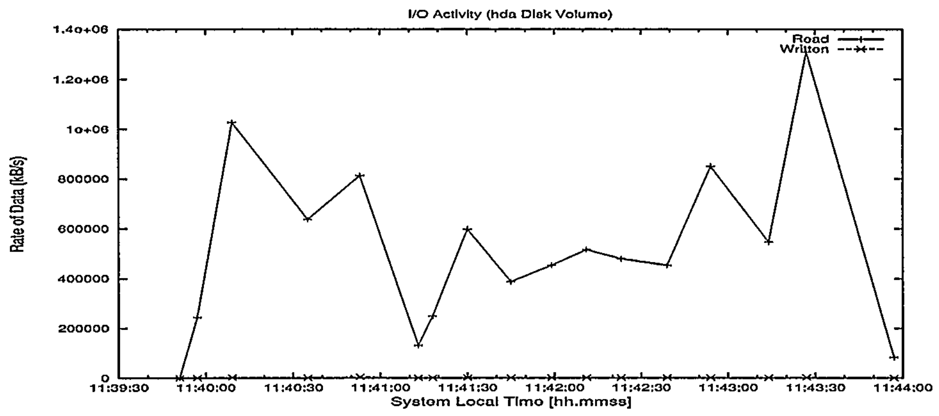


(b) Memory Usage

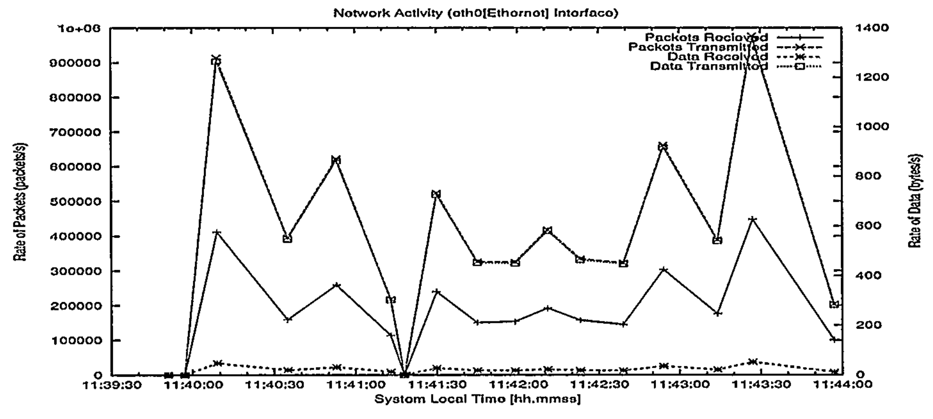


(c) Page Faulting

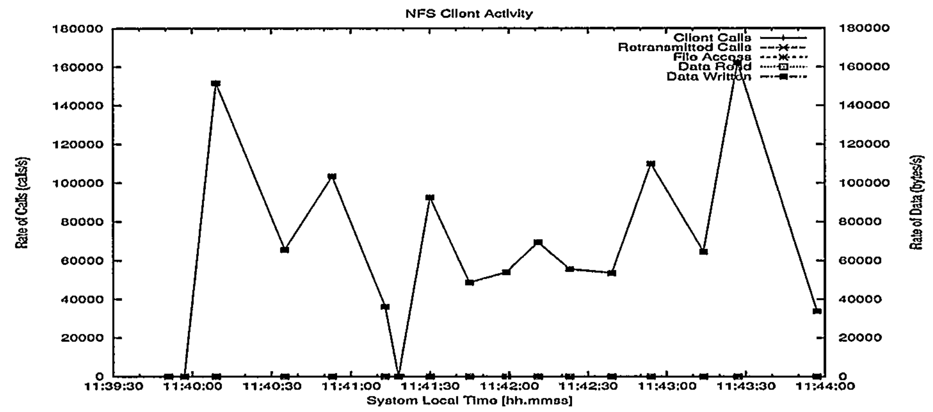
Figure 5.16: File I/O Job (Local Volume to Remote Volume)



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.17: File I/O Job (Local Volume to Remote Volume)

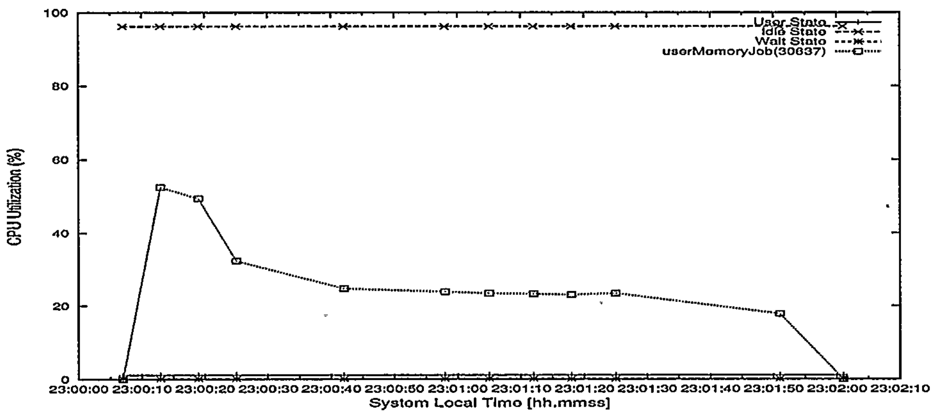
task on the same computer. Also, to identify the bottlenecks for the job, and how the performance of the job can be improved.

The design of this experiment is similar to that of Experiment 4; the difference is that here the competing task starts ahead of the job, and the competing task is a memory workload program. The competing task is started on the computer where the job was scheduled to run. The job is submitted at the point when the competing task has run long enough to use more than 50% of the available physical memory on the computer. Figures 5.18 and 5.19 show the results obtained in this experiment.

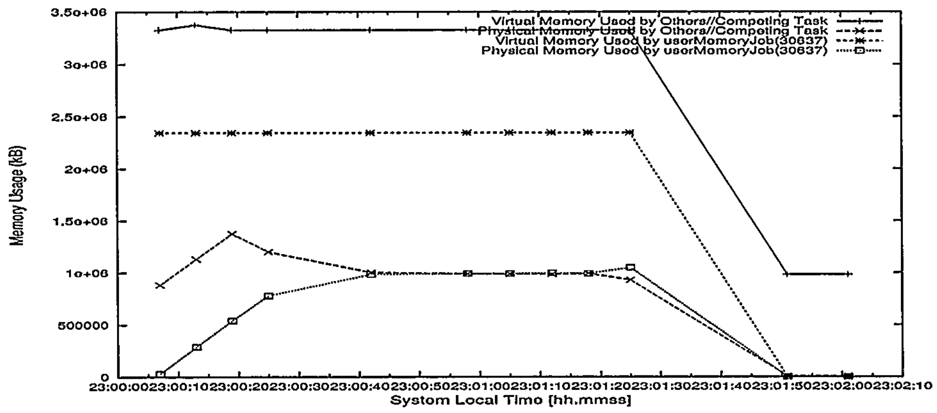
Figure 5.18(a) shows that the job received less than 50% CPU processing time at the beginning of the experiment. Comparing Figure 5.18(a) with Figure 5.6(a) (in Experiment 2) shows that the job in this experiment received 50% CPU processing time compared to the same job in Experiment 2. This result indicates that there is strong competition for CPU processing time between this job and some other competing tasks on the computer.

It can be observed in Figure 5.18(b) that the amount of physical memory used by both the job and the competing task increased between 23:00:00 and 23:00:20. Then, the physical memory used by the competing task decreased, while the physical memory used by the job increased until 23:00:40. At about 23:00:45, both the job and the competing task were using about the same amount of physical memory. This trend continued until 23:01:25 when the competing task disappeared. A close observation shows that the disappearance was due to the termination of the job and the competing task by the operating system.

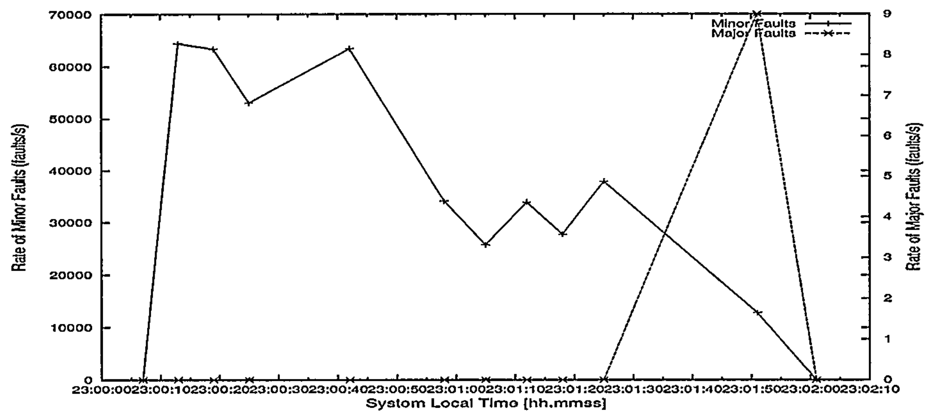
The page fault information in Figure 5.18(c) shows that the job started with some minor page faults then some major page faults at a point where the job and



(a) CPU Utilization

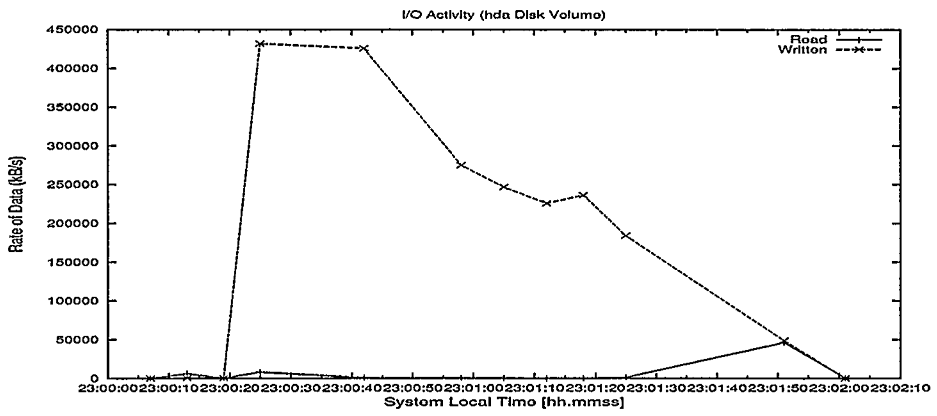


(b) Memory Usage

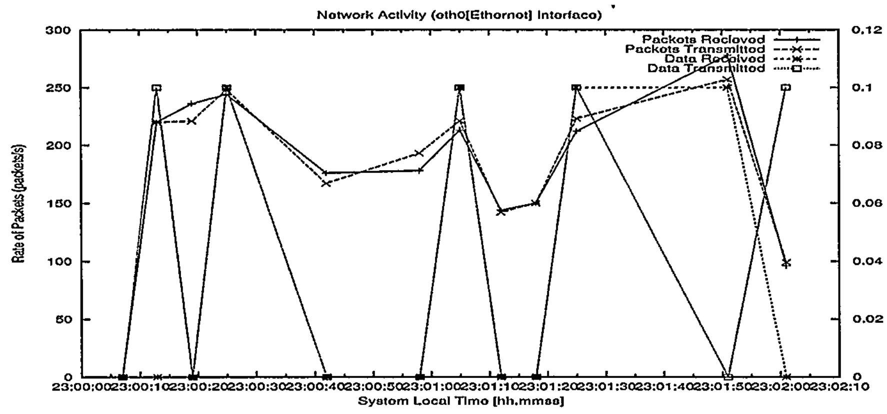


(c) Page Faulting

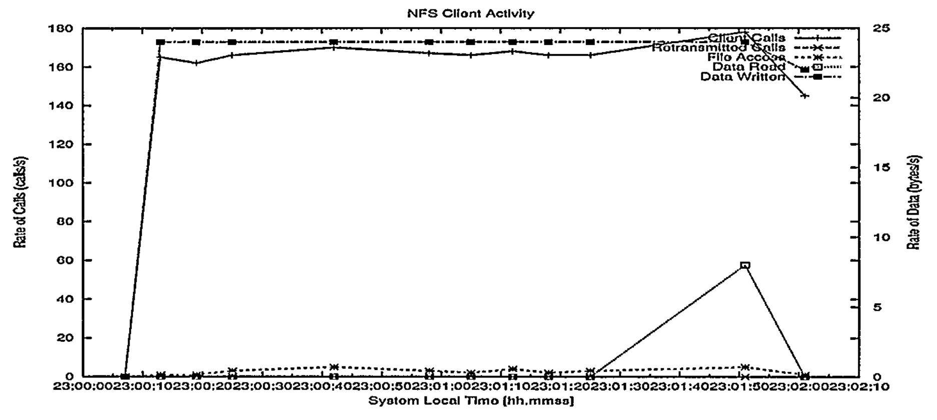
Figure 5.18: A Memory Workload Job Competing with Memory-Intensive Task



(a) Disk Input/Output Activity



(b) Network Activity



(c) Shared File System Activity

Figure 5.19: A Memory Workload Job Competing with Memory-Intensive Task

competing task were using the same amount of physical memory. The result shows that the number of faults per second is about half of what was observed in Experiment 2. This information indicates that there are tasks competing with the job on the computer. As a result, there is a strong competition for the physical memory on the computer.

Figure 5.19(a) shows that more data is written to disk and less data is read from disk compared to Figure 5.7(a) (in Experiment 2). There is no much difference in the network activity in this Experiment compared to that observed in Experiment 2 except the pattern; the same applies to the shared file system activity.

In this experiment, the monitoring information shows that there was an existing task competing with the job on the same computer. The monitoring information shows the competition and indicates that the job was not getting enough physical memory. It also showed that the job did not finish successfully.

The monitoring information generated from this experiment shows that the bottleneck for the job is physical memory. Identifying this bottleneck would help the user to make a decision on improving the performance of this job. In this situation, the user can submit the job to the same computer at a later time or submit the job to another computer with more physical memory resources. If these options do not help, the user may discuss the issue with the system administrator based on the monitoring information.

5.4 Validation

Most of the monitoring information reported in this thesis are retrieved from the *ps* (process status) utility tool except the page faults information. The page fault information is retrieved from *proc* (Process Information Pseudo-Filesystem) on UNIX operating system. The *ps* tool gets its results from *proc*; hence, all the information reported in this thesis are indirectly from *proc*.

UNIX tools like *vmstat*, *free*, *top*, and *sar* (a UNIX tool for collecting, reporting, and saving system activity information) could be used for validating the information reported by the monitoring tool developed in this thesis. The afore-mentioned tools also get their information from *proc* on a Linux system. In tests, the information reported using these tools is the same as that collected with the Wrapper based monitoring tool.

5.5 Summary

In this chapter, eight experiments were carried out to show how the monitoring information generated using the *Wrapper method* can help users to identify the bottlenecks to their jobs in grid computing environment.

The experiments were designed to show different aspects of a system including CPU utilization, memory usage, disk I/O, shared file system, and network. The monitoring tool was able to provide information about how the submitted jobs interacted with other processes on the system. The monitoring information from the experiments also showed how the use of one shared system resource is related to others. This was demonstrated in the network traffic and file system results.

The monitoring information shows if a job experienced competition for system resources, when the competition occurred, and how the competition affected the behaviour of the job. The monitoring information shows if a job is file I/O or memory-intensive. For file I/O intensive jobs, it shows the disk volume(s) where the data is read from and written onto.

In addition, the monitoring information showed that memory, local disk I/O, and shared file system can be bottlenecks to the performance of a job. Suggestions were made on how to improve the performance of jobs affected by these bottlenecks.

Chapter 6

Conclusion

This chapter summarizes the research work done for this thesis. The contributions of this thesis work are highlighted. It also provides suggestions of future research directions building upon this thesis work.

6.1 Thesis Summary

In a grid environment, a user does not know what is happening to a job when it is running on a computer. Monitoring jobs become essential so that users can see if their jobs have bottlenecks, and make decisions on how to improve the performance of their jobs.

There are various tools for monitoring the performance of grid infrastructure and grid applications. The tools give accurate information on basic system configuration, memory, and CPU usage statistics; but they do not give information about individual jobs. Recent tools that report on behavioural aspects of jobs have done so in cumbersome manners.

In this thesis, a technique for monitoring jobs in grid computing environment known as *Wrapper method* is designed, and a tool that implements the *Wrapper method* is also developed. The technique is called *Wrapper method* because a script called *Wrapper script* is used to monitor jobs. The *Wrapper script* monitors a job on whatever computer the job is scheduled to run. How the *Wrapper method* works is described in detail in Chapter 4.

The *Wrapper method* was implemented by developing a monitoring tool. The monitoring tool monitors a job on a computer and generates monitoring data on the performance of the job. A graphing tool is also developed to convert monitoring data into meaningful monitoring information. The monitoring information includes CPU utilization, memory usage, disk Input/Output, shared file system, and network information. The monitoring information is presented in time-series format in order to show the complete life cycle of a job. Chapter 4 describes how the *Wrapper method* is implemented in this thesis.

Eight experiments were conducted in this thesis to validate the job monitoring capabilities of the *Wrapper method.* The experiments were designed in such a way that they show how information from the monitoring system would help users to understand the behaviour of their jobs on a computer. In addition, suggestions were made on how to improve the performance of a job in each scenario.

In each experiment the *Wrapper script* is submitted, and it starts the user's job on a computer. The *Wrapper script* also starts the tool that monitors the job on the computer. The monitoring tool retrieves monitoring data about the job from the computer where the job is running. When the job is completed, the monitoring data is transformed into meaningful monitoring information by the graphing tool.

Two computing tasks were used as workloads in the experiments conducted in this thesis work. The tasks are computer programs that are designed to run on a computer. The first program uses a large amount of physical memory on a computer. The second program does file I/O using local disk, network, and shared file system resources. The first program is called memory workload and the second program is called file I/O workload. The workloads are described in detail in Chapter 5 of this thesis.

Each workload has two programs: the job and the competing task. The *Wrapper script* starts the job, and the competing task is started on the computer by the user. The job and the competing task used the same computer program, so neither the job nor the competing task had an undue advantage over each other. In some experiments, a computing task that uses a large amount of system resources is started on the same computer where the job is running in order to introduce competition for system resources.

The experiments were designed to show different aspects of a system including CPU utilization, memory usage, disk I/O, file system, and network. The monitoring tool provided information on how the submitted jobs interacted with other tasks on the computer. The monitoring information helped in understanding what happened to jobs on a computer under different scenarios. Users would be able to answer some questions about their jobs by looking at the monitoring information. For example, did their job get an expected amount of CPU utilization, did their job get enough memory resources, did their job use disk I/O effectively, did a task or tasks compete with their job for shared resources like network bandwidth and shared disk volume; if there is competition for resources, when did it happen and what is the impact on the user's job.

The monitoring information would help users to identify the bottlenecks to their job on a particular computer; this in turn would help them in making a decision on how to improve the performance of their job. For example, if physical memory is the bottleneck for a particular job, the user may choose to submit the job to another computer with more physical memory resources; or the user may submit his/her job to the same computer at a later time if the memory shortage was due to a competing task on the computer. In a situation where a job did not complete on time due to lots of disk I/O activity, the user may modify his/her job to use another shared file system. In a case where a shared file system is slow, the user may submit his/her job at a later time when the shared file system is less busy, or submit the job to another system with a faster shared file system.

The monitoring information can also help users to know what modification to make to their jobs so they can get a better performance on a particular computer in the future. The user might ask the grid computing site administrator what was happening on a particular system at a particular time.

6.2 Contributions

The following are the contributions of this thesis work:

– This thesis work provides a survey of the existing grid monitoring tools and methodologies; it analyzes them from job monitoring perspective. The survey identified their short comings and paved way for this research.

– Designed *Wrapper method* technique, for monitoring jobs in grid computing environments; this technique uses a simple script to monitor individual jobs in a shared-memory computing environment.

– Implemented the *Wrapper method* by developing a monitoring tool; the monitoring tool monitors a job on a computer and collects monitoring data about the job. A graphing tool is also implemented as part of this thesis work; the graphing tool converts monitoring data into meaningful monitoring information.

– Series of experiments were carried out in this thesis to validate the job monitoring capabilities of the *Wrapper method.* The results show that memory, local disk I/O, network, and shared file system can be bottlenecks to the performance of a job. Suggestions were made on how to improve the performance of jobs affected by these bottlenecks.

6.3 Future Work

Several directions remain open for further research in the area of monitoring jobs in grid computing environments. This thesis focused on monitoring jobs on a computer in a shared-memory environment. It would be useful to improve on the *Wrapper method* so it could monitor jobs running on parallel computers in a distributed-memory environment.

A distributed-memory environment refers to a multi-processor computer environment where an individual processor can only address part of the total address space. The processes in such an environment coordinate their computations and share data by sending/receiving messages over the network. Jobs that are designed to take advantage of parallelism can execute faster than their sequential counterparts. Although, there are two sources of overhead: it takes time to construct and send a message from one processor to another, and a receiving processor must be interrupted in order to deal with messages from other processors [20].

The challenge of monitoring a distributed-memory parallel job using the *Wrapper method* is due to the fact that a scheduler does not know how a parallel job is started.

Therefore, the monitoring tool will lose track of some processes that belong to the job since it does not know how to relate the processes to the job. It is possible to know the hosts on which the processes associated with a parallel job are being executed, but it is difficult to know exactly what processes belong to the parallel job in a situation where the user has other processes running on the same computer at the same time. Considering the benefits of parallel jobs in distributed-memory environment, it would be beneficial to extend the *Wrapper method* to monitor such jobs.

A "post-mortem" approach is used in presenting the monitoring information to users in this thesis. In this implementation, a user would know what happened to his/her job after it is completed. That is, a user cannot have access to the monitoring information until the job is finished. Some users may prefer to see the way their job is performing in real time, so they could notify the system administrator if their job is not doing well on a computer. Hence, it would be very helpful to extend the current implementation to show the behaviour of jobs in real time.

The current implementation of the *Wrapper method* retrieves monitoring data from the *proc* file system on Linux operating system. The *proc* file system is not portable across all operating system platforms. In addition, the structure of *proc* directory and the layout of data in some *proc* files are different from one variant of UNIX to another. Therefore, the future work would be to make the *Wrapper method* portable.

Finally, research should be carried out on automated analysis of monitoring data. This could detect when a problem has been encountered and inform the user. It could also make the monitoring information more accessible to non-expert users. In addition, the results of the automated analysis could be used for performance tuning and analysis.

Bibliography

[1] A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. Van Lingen, R. Mcclatchey, H. Newman, W. Rehman, C. Steenberg, M. Thomas, and I. Willers. Job Monitoring in an Interactive Grid Analysis Environment. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP 04)*, Interlaken, Switzerland, Sep 2004.

[2] A. Natrajan and M.P Walker. Monitoring Remote Jobs in a Grid System, Nov 2005. (Unpublished).

[3] A. Puliafito, O. Tomarchio, and L. Vita. MAP: Design and Implementation of a Mobile Agent Platform. *Journal of System Architecture*, 46(2):145–162, 2000.

[4] A. S Grimshaw and W. A Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), Jan 1997.

[5] A. Abbas. *Grid Computing Technology - An Overview.* Charles River Media, 2003.

[6] A. Abbas. Grids in Telecommunications Sector. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 19, pages 365–372. Charles River Media, 2004.

[7] A. Ali, A. Anjum, J. Bunn, R. Cavanaugh, F. van Lingen, R. McClatchey, H. Newman, W. Rehman, C. Steenberg, M. Thomas, and I. Willers. Job monitoring in an interactive grid analysis environment. In *Proceedings of Com-*

puting in High Energy and Nuclear Physics (CHEP) conference, Interlaken, Switzerland, 2004.

[8] B. Balis, M. Bubak, W. Funika, T. Szepieniec, R. Wismuller, and M. Radecki. Monitoring Grid Applications with Grid-Enabled OMIS Monitor. In *Proceedings of the European Across Grids Conference*, pages 230–239, Santiago de Compostela, Spain, Feb 2004.

[9] B. Bartosz, M. Bubak, W Funika, R. Wismuller, M. Radecki, T. Szepieniec, T. Arodz, and T. M. Kurdziel. Grid Environment for On-Line Application Monitoring and Performance Analysis. *Scientific Programming*, 12(4):239–264, 2004.

[10] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson. A Monitoring Sensor Management System for Grid Environments. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 97–104, Pittsburgh, Pennsylvania, USA, Aug 2000.

[11] B. Tierney, D. Gunter, J. Becla, B. Jacobsen, and D. Quarrie. Using NetLogger for Distributed Systems Performance Analysis of the BaBar Data Analysis System. In *Proceedings of Computers in High Energy Physics (CHEP 2000)*, feb 2000.

[12] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A Grid Monitoring Architecture. Technical report, Global Grid Forum GMA Working Group, Aug 2001. Revised 16-January-2002.

[13] Z. Balaton and G. Gombas. Resource and Job Monitoring in the Grid. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 404–411, Klagenfurt, Austria, 2003.

[14] Bandwidth Market. Bandwidth and internet access catalogs, nov 2005. Available online at www.bandwidthmarket.com/resources/glossary/J1.html.

[15] P. Burden. Processes, nov 2005. Available online at http://www.scit.wlv.ac.uk/ jphb/spos/notes/processes.html.

[16] ClimatePrediction.net. Climateprediction.net, nov 2005. Available online at http://www.climateprediction.net/index.php.

[17] Cluster Resources. Moab Workload Manager Administrator's Guide. Technical report, Cluster Resources, Inc., Jan 2006. version 4.5.0.

[18] Cluster Resources Incorporated. Maui Users Manual, nov 2007. Available online at http://www.clusterresources.com/products/maui/docs/mauiusers.shtml.

[19] N. Committee. *Realizing the Information Future: The Internet and Beyond.* National Academy Press, 1994.

[20] Computational Science Education Project. Distributed Memory, aug 2007. Available online at http://www.ipp.mpg.de/de/for/bereiche/stellarator/Comp_sci/CompScience/csep/csep1.phy.ornl.gov/ca/node21.html.

[21] Condor Project. Classified Advertisements, nov 2005. Available online at http://www.cs.wisc.edu/condor/classad/.

[22] Condor Project. Hawkeye, nov 2005. Available online at http://www.cs.wisc.edu/condor/hawkeye.

[23] J. Coomer and C. Chaubal. Introduction to the Cluster Grid - Part 1. Technical report, Sun Microsystems - Sun BluePrints OnLine, Aug 2002.

[24] CrossGrid. Developing new Grid components, Nov 2005. Available online at http://www.eu-crossgrid.org.

[25] D. Gunter and B. Tierney. Netlogger: A Toolkit for Distributed System Performance Tuning and Debugging. In *Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management, (IM 2003)*, Colorado Springs, CO, mar 2003.

[26] D. Mills. Simple Network Time Protocol (SNTP) RFC 1769. Technical report, University of Delaware, mar 1995. Available online at http://www.eecis.udel.edu/ ntp/.

[27] D. Thain, T. Tannenbaum, and and M. Livny. Condor and the Grid. In F. Berman and A. J.G. Hey and G. Fox, editor, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11, pages 299–336. John Wiley, 2003.

[28] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation - Practice and Experience*, 17(2), 2005.

[29] D.A. Reed and C.L.Mendes. Intelligent Monitoring for Adaptation in Grid Applications. In *Proceedings of the IEEE Special Issue onProgram Generation, Optimization, and Platform Adaptation*, Feb 2005.

[30] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R.Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, 15(4):327–344, Dec 2001.

[31] F. Sacerdoti, M. Katz, M. Massie and D. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of IEEE International Conference on Cluser Computing (Clusters 2003)*, Hong Kong, Dec 2003.

[32] D. J. Feldman. Managing grid environment. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 15, pages 295–308. Charles River Media, 2004.

[33] G. S. Fishman. *Monte Carlo Concepts, Algorithms and Applications.* Springer-Verlag, 1996.

[34] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proceedings of the International Conference on Network and Parallel Computing (IFIP)*, pages 2–13, Beijing, China, Nov 2005.

[35] I. Foster and C. Kesselman. *The Globus Toolkit.* Morgan Kaufmann, first edition, 1999.

[36] E. Frisch. *Essential System Administration.* O'Reilly, second edition, 1995.

[37] T. Garritano. Globus: An Infrastructure for Resource Sharing. *Clusterworld*, 1(1), Dec 2003.

[38] J. Gartner. Supercomputers Speed Car Design, April 2004. Available online at http://www.wired.com/news/autotech/0,2554,63185,00.html.

[39] M. R. Genesereth and S. P. Ketchpel. Software Agents. *Communications of the ACM*, 37(7), Jul 1994.

[40] M. Gerndt, R. Wismuller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorski, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. Performance Tools for the Grid: State of the Art and Future. APART White Paper 3-8322-2413-0, University of Technology Munich, 2004.

[41] GGF. About the Globus Toolkit, nov 2005. Available online at http://www.globus.org/toolkit/about.html.

[42] GGF. Globus, nov 2005. Available online at http://www.globus.org/.

[43] GGF. Globus Toolkit, nov 2005. Available online at http://www.globus.org/toolkit/.

[44] GGF. The Globus Alliance, nov 2005. Available online at http://www.globus.org/alliance/.

[45] GIMPS. Greater internet mersenne prime search, nov 2005. Available online at http://www.mersenne.org/prime.htm.

[46] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer-Verlag, 2003.

[47] Global Grid Forum GMA Working Group. Global Grid Forum GMA Working Group - Home Page, nov 2005. Available online at http://www-didc.lbl.gov/GGF-PERF/GMA-WG/.

[48] GRC. Background on Grid Computing, nov 2005. Available online at http://grid.ucalgary.ca/resources.html.

[49] GridLab. GridLab: A Grid Application Toolkit and Testbed, Nov 2005. Available online at http://www.gridlab.org.

[50] A. Grimshaw. Data Grids. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 8, pages –. Charles River Media, 2004.

[51] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proceedings of the IEEE Mascots 2000 Conference (Mascots 2000)*, pages 267–273, San Francisco, California, Aug 2000.

[52] H. Kejing, D. Shoubin, Z. Ling, and S. Binglin. Building Grid Monitoring System Based on Globus Toolkit: Architecture and Implementation. *Lecture Notes in Computer Science*, 3314:353–358, Jul 1994.

[53] H. Truong and T. Fahringer. SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid. *Scientific Programming*, 12(4):225–237, 2004.

[54] H.B. Newman, I.C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. Monalisa: A distributed Monitoring Service Architecture. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, California, March 2003.

[55] H.D. Karatza. Scheduling Parallel and Sequential Jobs in a Partitionable Parallel System. *International Journal of Simulation: Systems, Science Technology, UK Simulation Society*, 4(1 & 2), 2003.

[56] J. Hollingsworth and B. Tierny. *Instrumentation and Monitoring.* Morgan Kaufmann, second edition, 2004.

[57] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002, Nov 2005. Available online at citeseer.ist.psu.edu/foster02physiology.html.

[58] I. Foster, S. Tuecke, and C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), Jun 2001.

[59] IBM. Grid computing: What are the key components?, nov 2005. Available online at http://www-128.ibm.com/developerworks/grid/library/gr-overview/.

[60] IBM. Lessons learned from the TeraGrid, Part 3: Putting the pieces together, june 2006. Available online at http://www-128.ibm.com/developerworks/grid/library/gr-teragrid3/.

[61] J. Bunn, D. Bouriklov, R. Cavanaugh, I. Legrand, A. Muhammad, H. Newman, S. Singh, C. Steenberg, M. Thomas, and F. Van Lingen. A Grid Analysis Environment Service Architecture, Nov 2005. Available online at http://ultralight.caltech.edu/gaeweb/gae_services.pdf(Unpublished).

[62] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, San Francisco, USA, aug 2001.

[63] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, 2002.

[64] J. M. Schoff and B. Clifford. Monitoring Clusters and Grids. *Clusterworld*, 2(17), Aug 2004.

[65] B. Jacob. Grid computing: What are the key components?, nov 2005. Available online at http://www-128.ibm.com/developerworks/grid/library/gr-overview/index.html.

[66] B. Jacob, L. Ferreira, N. Bieberstein, C. Gilzean, J.-Y. Girard, R. Strachowski, and S. S. Yu. *Enabling Applications for Grid Computing with Globus.* IBM Press, first edition, 2003.

[67] D. Johnson. Desktop Grids. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 5, pages –. Charles River Media, 2004.

[68] J. Joseph and C. Fellenstein. *Grid Computing.* IBM Press, first edition, 2003.

[69] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems.

In *Proceedings of IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing*, Orlando, FL, Mar 1998.

[70] K. Rothermel and R.Popescu-Zeletin. Mobile Agents. In *1st Internation Workshop, MA'97*, Berlin, Germany, Apr 1997.

[71] D. Kalev. LINUX TIPS AND TRICKS - Raw Disk I/O , may 2006. Available online at http://www.itworld.com/nl/lnx_tip/10122001/.

[72] M. Laucelli and J. Masso. Grids in Telecommunications Sector. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 18, pages 351–364. Charles River Media, 2004.

[73] I. Lumb. Cluster Grids. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 6, pages –. Charles River Media, 2004.

[74] M. Arlitt, E. Anderson, R. Curry, and R. Simmonds. Using DataSeries with a Job Centric Monitoring Service. In *Proceedings of the 13th annual HP Open View University Association workshop*, Sophia Antipolis, France, may 2006.

[75] M. Cannataro, C. Mastroianni, D. Talia, and P. Trunfi. Evaluating and Enhancing the Use of the GridFTP Protocol for Efficient Data Transfer on the Grid. In *Proceedings of Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, Venice, Italy, Oct 2003.

[76] M. Gerndt R. Wismuller, Z. Balaton, G. Gombas, Z. Nemeth, N. Podhorski, H.L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. Performance Tools for the Grid: State of the Art and Future. Technical report,

Lehrstuhl fuer Rechnertechnik und Rechnerorganisation, Technische Universitaet Muenchen (LRR-TUM), Jan 2004.

[77] R. Martin. The God Particle and the Grid. *Wired Magazine*, April 2004.

[78] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, Jul 2004.

[79] N. Podhorszki, Z. Balaton, and G. Gombas. Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor. *Lecture Notes in Computer Science*, 3165:179–181, 2004.

[80] NETDICTIONARY. DARPANET, nov 2005. Available online at http://www.netdictionary.com/a.html.

[81] O. Tomarchio and L. Vita and and A. Puliafito. Active Monitoring in Grid Environments using Mobile Agent Technology. In *Proceedings of the 2nd Workshop on Active Middleware Services (AMS) at HPDC-9*, Pittsburg, USA, Aug 2000.

[82] M. Oberdorfer and J. Gutowski. Grids in Life Sciences. In D. Pallai, editor, *Grid Computing: A Practical Guide to Technology and Applications*, chapter 17, pages 341–350. Charles River Media, 2004.

[83] T. F. of SuperComputing. Glossary and acronym list, nov 2005. Available online at http://books.nap.edu/html/up_to_speed/appD.html.

[84] Pablo Group. Scalable Performance Tools (Pablo Toolkit), nov

2005. Available online at http://vibes.cs.uiuc.edu/Project/Pablo/ScalPerfTools/Overview.htm.

[85] PBS Pro. PBS Pro 5.1 User Guide, nov 2005. Available Online at http://www.raunvis.hi.is/ finnboo/bjolfur/pbs_user_guide.pdf.

[86] G. Pfister. *In Search of Clusters.* Prentice Hall PTR, second edition, 1998.

[87] R. Curry and R. Simmonds. Job Centric Cluster Monitoring. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, Minneapolis, USA, july 2006.

[88] Red Hat Inc. *Red Hat Linux System Administration Primer.* Red Hat Inc., 2003.

[89] Rik van Riel. proc(5) - Linux man page, nov 2007. Available online at http://linux.die.net/man/5/proc.

[90] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: adaptive control of distributed applications. In *Proceedings of the Seventh IEEE Symposium on High-Performance Distributed Computing*, pages 172–179, Chicago, Illinois, Jul 1998.

[91] R. Y. Rubinstein. *Simulation and the Monte Carlo Method.* Wiley, 1981.

[92] R.Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computer Systems*, 15(5-6), 1999.

[93] S. Zanikolas and R. Sakellariou. A Taxonomy of Grid Monitoring Systems. *Future Generation Computer Systems*, 21(1), 2005.

[94] SETI@home. SETI@home, nov 2005. Available online at http://setiathome.ssl.berkeley.edu/.

[95] E. Shaffer, D. A. Reed, S. Whitmore, and B. Schaeffer. Virtue: Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, 32(12):44–51, Dec 1999.

[96] L. Smarr. Grids in Context. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 1, pages 3–12. Morgan Kaufmann, second edition, 2004.

[97] W. R. Stevens. Addison-Wesley, addison-wesley professional computing series edition, 1993.

[98] Sun Microsystems. XDR: External Data Representation Standard. Technical Report IETF RFC 1014, , Jun 1987. Available online at www.lpds.sztaki.hu/publications/reports/ lpds-2-2000.pdf.

[99] T. Shaun. *Enterprise JMS Programming.* Wiley, New York, 2002.

[100] The Globus Alliance. GT 3.9.3 WS_GRAM Approach, june 2006. Available online at http://www-unix.globus.org/toolkit/docs /development/3.9.3/execution/wsgram/WS_GRAM_Approach.html.

[101] The Globus Alliance. Cluster Resources Incorporated, nov 2007. Available online at http://www.globus.org/api/c-globus-2.2/globus_gram_documentation/html/index.html.

[102] V. Quma and R. Lachaize and E. Cecchet. An asynchronous middleware for

Grid resource monitoring. *Concurrency and Computation - Practice and Experience*, 16(5):523–534, 2004.

[103] R. van Riel. Manpage of PROC, nov 2005. Available online at http://www.die.net/doc/linux/man/man5/proc.5.html.

[104] J. Vetter and D. A. Reed. Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids. *Journal of High Performance Computing Applications*, 14(4):357–366, 2000.

[105] M. M. Waldrop. Grid Computing, nov 2005. Available online at http://www.technologyreview.com/articles/02/05/waldrop0502.asp?p=1.

[106] WIKIPEDIA. Definition of Distributed Computing, nov 2005. Available online at http://en.wikipedia.org/wiki/Distributed_computing.

[107] WIKIPEDIA. File system, jul 2006. Available online at http://en.wikipedia.org/wiki/File_system.

[108] Wikipedia. Page fault, nov 2007. Available online at http://en.wikipedia.org/wiki/Page_fault.

[109] Yuan Cangzhou. Chapter 3 Memory Management Virtual Memory System, nov 2007. www.opentech.com.cn/knowledge_center/ courseware/manageCourseware.php?action=downloadPresentation&id=15.

[110] Z. Balaton, P. Kacsuk, N. Podhorszki, and F. Vajda. Comparison of Representative Grid Monitoring Tools. Technical Report LDS-2/2000, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 2000. Available online at www.lpds.sztaki.hu/publications/reports/lpds-2-2000.pdf.

[111] Z. Balaton, P. Kacsuk, N. Podhorszki, and F. Vajda. From Cluster Monitoring to Grid Monitoring based on GRM. In *Proceedings of the EuroPar' 2001*, pages 874–881, Manchester, UK, Aug 2001.

Appendix A

Glossary

BSD (Berkeley Software Distribution) refers to the particular version of the UNIX operating system that was developed at the University of California at Berkeley

SysV (System V) was one of the versions of the UNIX computer operating system. It was originally developed by AT&T and first released in 1983.

POSIX is an acronym for Portable Operating System Interface. It is a standard to allow applications to be source-code portable from one system to another. POSIX consist of several separate standards corresponding to different parts of a computer system.

Process Wait Channel (WCHAN) is the address of an event on which a particular process is waiting.

Physical Memory is the memory hardware (normally Random Access Memory) installed on a system.

Logical memory as opposed to physical memory is the way memory is organized by the operating system. In order to use the physical memory of a computer, such as RAM chips or cache, the operating system organizes the memory into some logical manner, such as memory address.

Thrashing is a problem as a result of paging when there is not enough memory on the system for all the processes currently running. Therefore, the same pages are being loaded repeatedly due to a lack of enough physical memory to keep them in memory.

Memory Page is the smallest unit of memory handled by the operating system; the size of a memory page is usually 4 or 16 kilobytes.

Paging is a process by which memory pages are moved between a disk volume and the physical memory. The disk volume can be a local disk volume, a shared file system, or a swap device. Paging occurs in order to free up memory needed by a process.

Swapping is similar to paging but swapping refers to writing an entire process to disk thereby freeing all of its memory.

Swap Space is used to describe a disk space used by the operating system kernel as "virtual" RAM to hold pages of data that have not been recently used and which no longer fit into paging space.

Virtual Memory is a memory management technique used by the operating system, where the disk is used as an extension of RAM so that the effective size of usable memory grows correspondingly. The part of the hard disk that is used as virtual memory is called the swap space.

Memory Resident Pages are pages that are permanently in the memory.

Appendix B

Monitoring Data Statistics

B.1 CPU Utilization Statistics

The first report generated by the iostat command is the CPU utilization report. For multiprocessor systems, the CPU values are global averages among all processors. The CPU utilization report has the following statistics:

%user: the percentage of CPU utilization that occurred while executing at the user level (application).

%nice: the percentage of CPU utilization that occurred while executing at the user level with nice priority.

%system: the percentage of CPU utilization that occurred while executing at the system level (kernel).

%iowait: the percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request.

%idle: the percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request.

B.2 Network Statistics

Bytes received/sent: These network interface statistics provides an indication of the bandwidth utilization of the network.

Interface counts and rates: These statistics can give indications of excessive collisions, transmit, and receive errors. These statistics (particularly if the statistics are available for more than one system on the network) helps in network troubleshooting.

Transfers per Second: Normally collected for block I/O devices, such as disk and high-performance tape drives, this statistic is a good way of determining whether a particular device's bandwidth limit is being reached. Due to the electromechanical nature of disk and tape drives, their performance rapidly degrades as their I/O limits are reached.

Packet Counts: The number of packets received and transmitted through a local or remote port gives an idea of the kind of network activity happening on a system. The packet counts shows if lots of communication is happening between the host system running a job and external systems or programs.

B.3 Memory Statistics

Page Ins/Page Outs: These statistics make it possible to gauge the flow of memory pages from the physical memory to the hard disk. High rates for both of these statistics can mean that the system is short of physical memory and is thrashing.

Active/Inactive Pages: These statistics show how heavily memory resident pages are used. A lack of inactive pages can point toward a shortage of physical memory.

Free, Shared, Buffered, and Cached Pages: These statistics provide additional detail over the active/inactive page statistics. By using these statistics, it is possible to determine the overall memory utilization in detail.

Swap Ins/Swap Outs: These statistics show the system's overall swapping behaviour. Excessive rates of these statistics can point to physical memory shortages.

Virtual Memory: This indicates the total number of uniquely-addressable memory space required by a program.

Resident Set Size: Resident set size is the aggregate size of the valid (that is, memory-resident) pages in the address space of a process. In a virtual memory system, a process' resident set is that part of a process' address space which is currently in the physical memory.

B.4 Disk Space Usage Statistics

Free Space: It is the amount of unused space on a system.

Transfers per Second: These statistics determines whether a particular device's bandwidth limitations are being reached.

Reads/Writes per Second: These statistics is a more detailed breakdown of the transfers per second statistics; it allows the system administrator to fully understand the nature of the I/O load on a storage device.

Appendix C

The Life Cycle of a UNIX Process

UNIX executes most kernel services within a process' context, by implementing a mechanism which separates the two possible modes of execution of a process. Hence, the unique "Running" state must be split into a "User Running" state and a "Kernel Running" state [15].

From Figure C.1, a process can be in any of the following distinct nine states:

Created: This is the state of a freshly created process. Whether freshly created processes are entirely resident in memory depends on the details of the memory management system. This state may also include processes that have not yet been fully created.

Ready to run, in memory: There is no reason why the process should not run apart from the fact that some other process is currently running.

Running in kernel mode: The process is running in kernel mode. It may be handling a system call or an interrupt or some other process (also in kernel mode) may have scheduled it to run. The process may determine that it has finished (either normally via an exit() or via some kernel detected abnormal condition) or that it is blocked awaiting some event such as a time signal or peripheral activity.

Running in user mode: This is the normal state of a process.

Pre-empted: The process has been interrupted and is about to resume normal user mode operation. The kernel scheduler may move a process into this state.

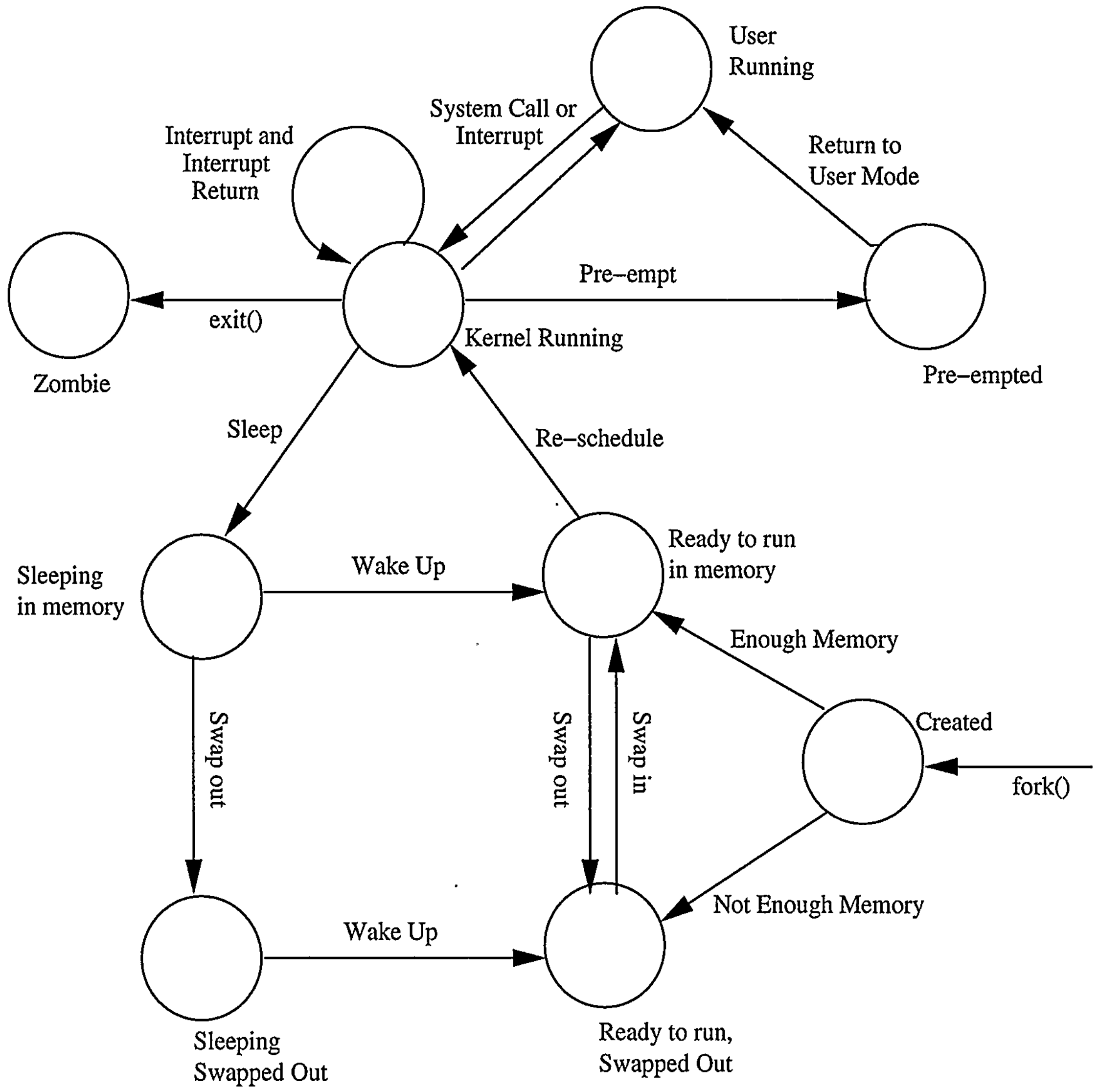


Figure C.1: Process Transition Diagram (Source [15])

Zombie (or defunct): The process will not run again, but information such as the exit code has not been collected by the parent process.

Sleeping in memory: The process is blocked awaiting an event. All that can happen is that the process can be woken up (by changing its status to "ready to run") or swapped out.

Sleeping, swapped out: The process is waiting for an event and has been swapped out.

Ready to run, swapped out: Before running the process needs to be copied back into memory.

Appendix D

UNIX System Tools

free gives a concise, simple overview of system memory and swap utilization.

vmstat is similar to free, but shows more information in addition to memory utilization statistics. It gives an overview of process, memory, swap, I/O, system, and CPU activity in one line of numbers.

top displays CPU utilization, memory utilization, and process statistics. Unlike the free command, top's default behaviour is to run continuously.

iostat displays an overview of CPU utilization, along with I/O statistics for one or more disk drives.

mpstat displays more in-depth CPU statistics.

sadc is known as the system activity data collector; sadc collects system resource utilization information and writes it to a file.

sar produces reports from the files created by sadc. sar reports can be generated interactively or written to a file for more intensive analysis.

lsof lists all the files opened by processes on the system. An open file may be a regular file, directory, block special file, character special file, executing text reference, library, stream, or network file (i.e., Internet socket, NFS file, or UNIX domain socket).

uptime gives a rough estimate of the system load; it reports the current time, the amount of time the system has been up, and three load averages; it reports the load averages for the past 1, 5, and 15 minutes.

df produces a report that describes all the filesystems, their total capacities, and the amount of free space available on each filesystem.

du reports the amount of disk space used by all files and subdirectories underneath one or more specified directories, listed on a per-subdirectory basis.

quot breaks down disk space usage within a single filesystem by user.

netstat is used to monitor a system's TCP/IP network activity. It provides some basic data about how much and what kind of network activity is happening on a system. It lists all the active connections with the local host. The number of data transferred between two systems via each connection is also reported in packets.

tcpdump allows the user to examine the headers of packets transmitted via TCP/IP.