MGST

McMaster Grid Scheduling Testing Environment

By
Majd Kokaly, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.A.Sc
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE (2008)	McMaster University Hamilton, Ontario

TITLE:	McMaster Grid Scheduling Testing Environment
AUTHOR:	Majd Kokaly, B.Sc.(Birzeit University)
SUPERVISOR:	Dr. Douglas G. Down
NUMBER OF PAGES:	xiii, 122

Abstract

With the phenomenal growth of the Internet and the advancement of computing hardware, grid architectures have been developed to exploit idle cycles in large networks of computational resources. One key aim of resource management (scheduling) schemes is to find mappings of incoming workload to machines within the grid to maximize the output. The first contribution of this thesis is the construction of a tool to aid researchers in testing and improving scheduling schemes, namely the McMaster Grid Scheduling Testing Environment (MGST).

The Linear Programming Based Affinity Scheduling Scheme (LPAS_DG) was introduced by researchers at McMaster, and simulation results have been promising in suggesting that this scheduling scheme outperforms other schemes when there is high system heterogeneity and is competitive under lower levels of heterogeneity. The second contribution of this research is providing suggestions to improve this scheme, based on the results of experiments where the LPAS_DG scheme was actually deployed on the MGST testbed.

Acknowledgements

The author would first like to acknowledge the guidance and great support provided by his supervisor Dr. Douglas Down over the past two years. The support of his wife and family is also appreciated. Issam Al-azzoni provided excellent feedback and suggestions during this research and hence the author would like to thank him. The author would like to thank Derek Lipiec as well for his help in setting up for the experiments conducted during this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Motivation

With knowledge comes the drive to pursue more knowledge. Recently, scientists have developed the need for huge computational power. This need along with the Internet coupled with the advancement of computers, has led to the development of grid technology.

Research areas and applications that require large computational power include biology, medicine, artificial intelligence, mathematics, cryptography and climate modelling. For instance, current DNA-based research requires huge computational power.

The introduction of the personal computer and later the advancement and spread of personal computers has contributed to the development of desktop grids. The personal computers sold today are more than five orders of magnitude faster than computers from 50 years ago [30].

The increase of the number of hosts connected to the Internet in recent years has been phenomenal. From 1993 to 2007, the number of hosts connected to the Internet increased by 19540% [27]. The majority of personal computers connected to the Internet spend most of their time idle. Harvesting the idle cycles of personal computers connected to the Internet can produce a powerful computing resource at low cost.

Grid technology is a powerful computational resource, and maximizing the output of grid systems is challenging. Deploying an efficient scheduling scheme to map jobs to machines is a key to maximize the output of a grid system.

The focus of this thesis is on grid architectures with heterogeneous processors. Processor heterogeneity may be caused by several factors. The first is the introduction of multi-core processors. Examples include the Cell processor used in the PLAYSTATION 3 and the Core 2 family manufactured by Intel. Some of these cores are non-identical, which results in heterogeneity. Some cores might be better in a particular type of computing (e.g. vector operations) and worse in another type. A second factor is the wide range of computing devices. Video game consoles as well as cellphones and Internet tablets are joining personal computers in connecting to the Internet. In the future, many devices that have a processor might be able to participate in grid architectures.

Heterogeneity can be exploited by scheduling schemes. One way of doing so is to send jobs to a server that can complete the job fastest. However, scheduling for heterogeneous grids is challenging as sending jobs to processors that execute that type of job slowly may result in wasting processing time which could have been used to execute different type of jobs efficiently, this in turn can harm the scheduling performance. In other words, because processors are different, choosing the right processor has a more significant effect on the performance of scheduling schemes than when processors are homogeneous (We will see this in more detail in Chapter 7).

Our work in this thesis is about testing and scheduling schemes, especially those for heterogeneous grids. This thesis work involves creating a testing environment to test and improve proposed policies.

1.2 Research Objective

In this thesis we will pursue the following research objective: *Provide a testing environment for theoretical scheduling policies on real grids.*

The testing environment should be able to give testers the ability to simulate a heterogeneous grid in the case that homogeneous servers are being used. In addition, we aim to develop an extensible environment to allow testers to add new scheduling schemes.

1.3 Contributions

The main contributions of this research are:

- The development of an extensible testing environment (**M**cMaster **G**rid **S**cheduling **T**esting Environment (**MGST**)) that makes it possible to test and improve scheduling schemes.

- Running experiments to test the implementability of the LPAS_DG scheduling scheme (defined in Chapter 4) and making suggestions to improve it.

1.4 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 introduces Desktop grids and Xgrid Technology. First desktop grids are discussed, followed by a brief explanation of Xgrid Technology. Finally, the future of desktop grids is touched upon.

Chapter 3 elucidates firstly the workload model followed in this research and then the machine availability model.

Chapter 4 serves as a literature review on scheduling schemes. In this chapter a taxonomy of desktop grids as well as a taxonomy of scheduling policies are presented. This is followed by a brief explanation of scheduling schemes used in this research.

Chapter 5 clarifies the system requirements specification and then explicates the software design.

Chapter 6 introduces background information for concepts used in the software implementation. This is followed by a brief explanation of the software packages and how to extend the software. This chapter is concluded with a discussion of considerations taken in the design and development phases.

Chapter 7 illustrates tests and results obtained using the testing environment developed.

Chapter 8 is the concluding chapter. It includes a discussion of the testing environment as well as the testing results followed by suggestions for possible future work.

Appendix A is a compact disc containing the source code and the executable for the testing environment in addition to Javadoc Documentation.

Appendix B serves as a user manual explaining the functions of the system and how to use them.

Appendix C is a collection of setting configurations procedures to help testers in using the testing environment for future tests.

Chapter 2

Desktop Grids

2.1 Desktop Grids

A desktop grid is a distributed computer system. The purpose of desktop grids is to provide specific computational or storage resources. The scale of such systems can be as small as a lab in a university campus or as large as the Internet itself. In [31], desktop grids are described as a large virtual computer formed by a networked set of heterogeneous machines that contribute with their resources. The main purpose of these systems is to exploit the dead cycles of millions of machines across the Internet [21].

Desktop grids are constructed from a number of machines and a Resource Management System (RMS). The machines (which from this point we will interchangeably call servers or machines) provide the computational and storage resources for the system. The functionality provided by an RMS varies depends on the type of desktop grid system. However, the basic service that any RMS will provide is accepting requests (jobs) from clients and mapping specific machines' resources to these requests. The RMS is central to the operation of desktop grid systems [31].

Currently desktop grids are used mainly for research purposes by different universities and research centres. Some grid based projects are discussed later in this chapter.

Desktop grid technology generates huge computational power that researchers can use to conduct computationally intensive experiments at reasonable cost. Other similar distributed computing technologies are clustered systems or dedicated grid systems, where the servers are owned and managed by one organization and used as a propriety system. Both technologies provide high computational power. Clustered systems are easier to manage and operate but usually more expensive than desktop grids since machines are bought and maintained at an organization's expense. On the contrary, in distributed desktop grids computational power is obtained by harnessing the idle cycles of voluntarily participating servers. Thus, a large amount of computational power can be obtained from a distributed desktop grid at reasonable cost, of course at the expense of more complex system management. Another important difference between desktop grids and other distributed computing technologies is the dynamic nature of servers in desktop grids. Servers can connect to the grid and disconnect at any time, making it harder to predict the availability of servers in such systems.

The invention and later the growth of this technology was driven by several factors. The first one is the existence of a suitable infrastructure. This infrastructure is constructed from the Internet and the hosts connected to it. The Internet provides a means of communication and the computers connected to it provide computational resources, meaning that building a Desktop grid requires only building a software layer on top of an existing system.

The second factor is the massive growth of the Internet and the fact that there are many connected hosts that are mostly idle. Figure 2.1 shows the exponential growth of hosts connected to the Internet. In January of 1993 only 2,217,000 hosts were discovered by the Internet Systems Consortium (ISC) survey host count, whereas in January of 2007 the survey discovered 433,193,199, an increase of 19540% [27].

The number of users connected to the Internet is estimated to be 1.3 billion, which means that around one fifth of the world's population uses the Internet [27]. That large number of users implies a correspondingly large number of connected computers. The 433,193,199 servers discovered by the ISC in January 2007 were servers directly

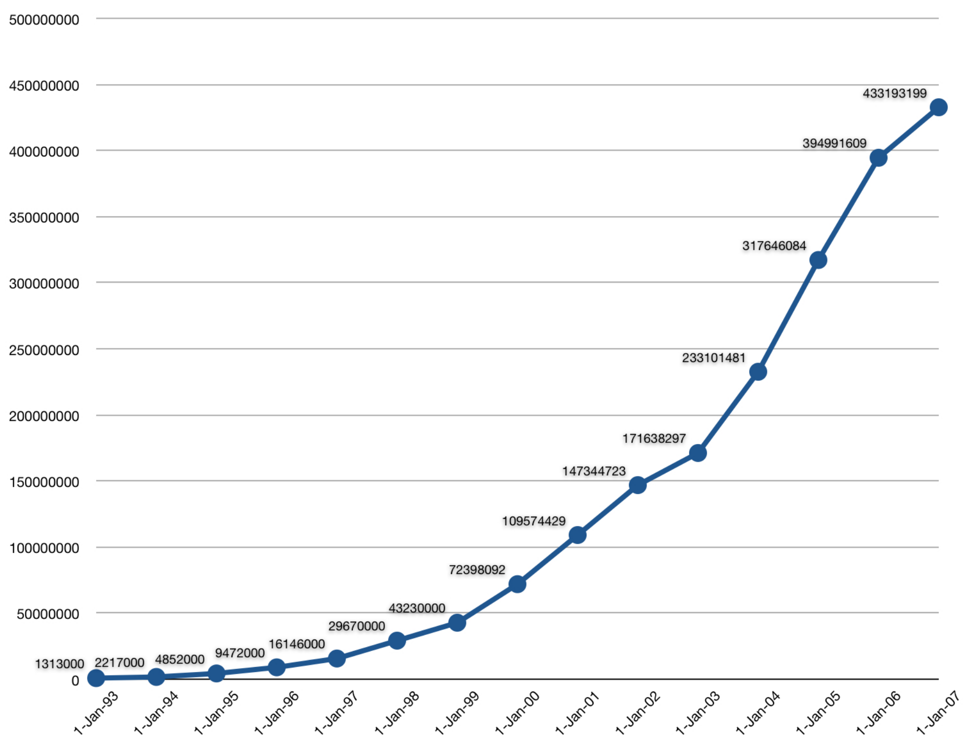


Figure 2.1: The exponential growth of Internet users

connected to the Internet (computers with real IP addresses) [28]. The actual number of hosts connecting through proxies is hard to measure, due to the fact that they are protected by firewalls. In any case, this large number of computers generates a huge potential of unexploited computing power.

The third factor is the exponential growth of computing power of individual computers. According to Moore's law, the number of transistors that can be inexpensively placed on an integrated circuit (IC) increases exponentially, doubling approximately every year [35]. Practically, the number of transistors placed on an IC circuit of the same size is doubled every 18 - 24 months (Figure 2.2). The increase in numbers of transistors results in a corresponding increase in computational operations done per second. It is worth mentioning that the introduction of multi-core processors (e.g. Intel Core 2 Duo) has significantly increased the computational power of personal computers. Multi-core processors also provide the ability to concurrently execute multiple threads. This ability allows for the execution of local tasks along with desktop grid tasks without affecting the performance of the local machine.

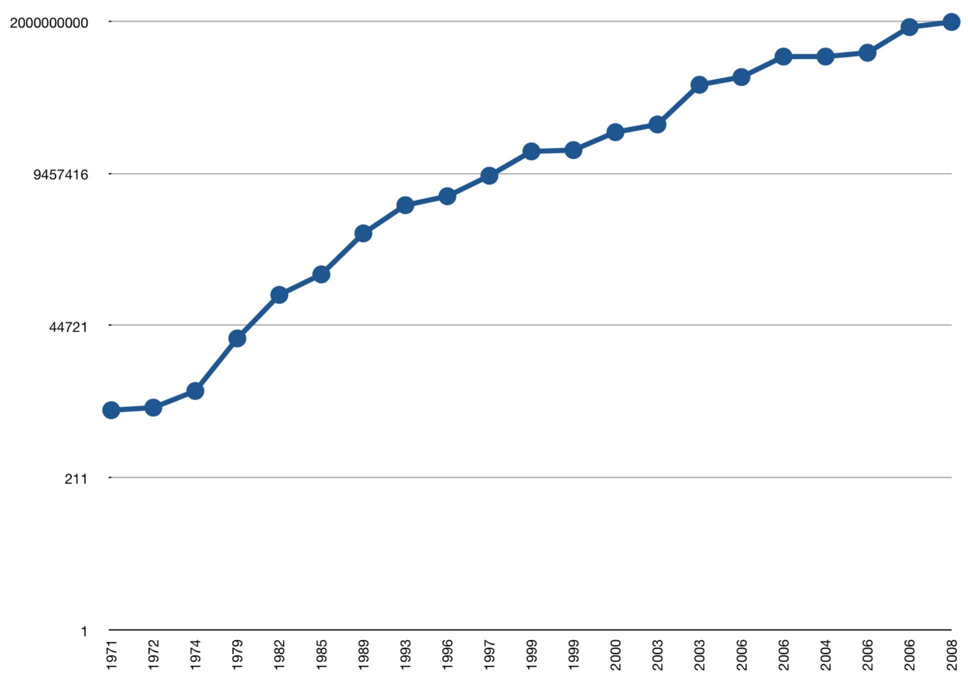


Figure 2.2: The exponential growth of the number of transistors per IC

In summary, desktop grids are computer systems developed to use the idle computing power of large numbers of computing machines. The widespread availability of low-cost, high performance computing hardware and the phenomenal growth of the Internet have created a suitable environment for desktop grid technology to be deployed.

2.2 Desktop Grids in Practice

2.2.1 History of Desktop Grids

Although there is no consensus about the origins of grid computing, the roots of this technology can be traced back to the late 1980s in fields related to distributed supercomputing for numerically intensive applications, with particular emphasis on scheduling algorithms (e.g. Condor [16], Load Sharing Facility [44]) [33]. Ian Forster and Carl Kesselman are known to be amongst the first scientists to write about the topic. Their publications include the seminal book *The Grid: Blueprint for a New Computing Infrastructure*, which was published in 1999 [21], and an important paper called *Physiology of the Grid*, which they co-wrote with several scientists in 2002 [22]. In addition to these publications, several projects are considered to be important milestones. Two of these projects are discussed in the following paragraphs.

Distributed.net is the first Internet distributed computing project [17]. It was founded in 1997. Distributed.net is a non-profit organization that tries to employ the computational power donated by thousands of its users around the world for academic research and public-interest projects. Their first project was RSA Secret-Key Challenge, which was initiated by the RSA Laboratories Institute (www.rsa.com) to demonstrate the relative security of different encryption algorithms. The challenge was successfully completed after 212 days and the RC5-56 encryption algorithm was cracked. Since then several projects have been successfully completed, while some are still active at the time of writing of this thesis. For example, the Optimal Golomb Rulers (OGR-25) project is still in progress. This project aims to find a solution for a mathematical problem called the Golomb Ruler, and has been active since September 2000 [18].

SETI@Home is another distributed computing project that uses Internet-connected computers. This project is managed by the Space Sciences Laboratory at the University of California, Berkeley. The group working on this project describes themselves on their web site as follows [41]:

> *SETI@home is a scientific experiment that uses Internet-connected com-*

puters in the Search for Extraterrestrial Intelligence (SETI). You can participate by running a free program that downloads and analyzes radio telescope data.

The SETI project was released to the public in May 1999. Although the project has not yet achieved its goal, it has proven the viability and practicality of the distributed computing concept. Another important goal was achieved when Berkeley Open Infrastructure for Network Computing (BOINC) was developed to support SETI@home and later turned into open source middleware for distributed computing. It is now being used in several distributed computing projects including Proteins@home and Rosetta@home. The list of desktop grids provided in the next section includes some of BOINC projects.

2.2.2 Examples of Desktop Grids

As mentioned earlier, the Internet's rapid spread and the large increase of computational power has resulted in an increased use of distributed grid computing. There are currently many desktop grids in operation. The following is a partial list.

- Proteins@home: Attempts to deduce the DNA sequence of given proteins [38].
- Rosetta@home: Tests the assembly of specific proteins, using appropriate fragments of better-known proteins [40].
- FightAIDS@home: Helps to identify candidate drugs that might have the right shape and chemical characteristics to block HIV protease [20].
- Compute Against Cancer: Used in cancer research [15].
- Artificial Intelligence System: Attempts to create a full simulation of the human brain [2].
- ABC@Home: Attempts to solve the ABC conjecture in Mathematics [1].

- SHA-1 Collision Search: This project investigates the characteristics of SHA-1 hashing algorithms in terms of collision properties [42].
- Project Sudoku: Searches for the smallest possible start configuration of a Sudoku game [45].
- APS@Home: Conducts research into the effects of atmospheric dispersion as it relates to the accuracy of measurements used in climate prediction [9].
- Spinhenge@Home: Models the spin of elementary particles in atoms using the principles of quantum mechanics [46].
- μFluids@Home: Simulates two-phase flow in microgravity and microfluidics problems [48].
- BURP: Aims to develop a publicly distributed system for rendering 3D animations. The BURP project is still in its Alpha stage. The public nature of this project makes it interesting and unique since users can upload animations to the grid to be rendered (i.e. request tasks from the grid) [12].
- SETI@home: As mentioned above, this project searches for extraterrestrial intelligence [41].
- Storage@home: In [11] the authors describe Storage@home as follows:

  > *Storage@home is a distributed storage infrastructure developed to solve the problem of backing up and sharing petabytes of scientific results using a distributed model of volunteer managed hosts. Data is maintained by a mixture of replication and monitoring, with repairs done as needed. By the time of publication, the system should be out of testing, in use, and available for volunteer participation*

  Storage@home is interesting because the main purpose behind it is not computing but storage.

Each desktop grid works on at most a few tasks or experiments at a time. These span several fields and topics such as Biology, Medicine, Artificial Intelligence, Mathematics, Cryptography, AI-based Games (e.g. Chess and Sudoku), Earth Sciences, Physics, and Astronomy. Biology, Mathematics and Cryptography seem to be the most active fields.

In spite of the fact that this type of research is relatively new and that the accomplishment of a task could take many years, a lot of tasks have been successfully completed. For instance, the PiHex Project found that the five trillionth bit of π is 0 [37]. Although the impact of this result on science is questionable, it is an indicator of the great computational power that grid computing can provide. On the other hand, other projects have larger impact such as the Genome Comparison Project which constructed a database comparing the genes from many genomes [24].

2.3 Xgrid Technology

In this section, the Xgrid Technology is discussed. Most of this section's material is taken from the official Xgrid manual [8].

2.3.1 Overview

Xgrid Technology is an Apple technology that simplifies the management and administration of distributed computing systems. Apple describes Xgrid in [8] as follows:

> *Xgrid, a technology in Mac OS X Server and Mac OS X, simplifies deployment and management of computational grids. Xgrid enables administrators to group computers into grids or clusters, and allows users to easily submit complex computations to groups of computers (local, remote, or both), as either an ad hoc grid or a centrally managed cluster.*

2.3.2 Xgrid Terminology

In Xgrid technology specific terms for its components and operations are used. The following are needed for this thesis:

- Grid: a group of computers that can collaboratively complete a job using the Xgrid technology in Mac OS X Server and Mac OS X.
- Controller: an Xgrid controller manages the grid and its work. It is built into Mac OS X Server.
- Agent: an Xgrid agent resides on one computer in a grid and runs tasks sent to it by the controller. Any computer running Mac OS X v10.3 or v10.4 can run an Xgrid agent.
- Task: a part of a job that one agent in the grid performs at one time.
- Client: any computer running Mac OS X v10.4 or Mac OS X Server v10.4 that submits a job to an Xgrid controller.
- Job: a set of work submitted to a grid from the client to the controller.

It is worth mentioning here that the Xgrid terminology is different than the terminology we use for our system. For example, there is no concept of a task in our system. In any case, the terminology and the way our system works is discussed in Chapter 5.

2.3.3 Xgrid Usage

Xgrid can be used for three variations of distributed computing. It can be used in clusters, distributed grids and local grids.

Xgrid Clusters are grids constructed from servers entirely dedicated to computation. Typically, cluster systems are collocated in a rack and connected via high performance networks. Also these servers are often completely homogeneous. This means that they have identical operating systems that run on similar hardware. These types of systems are managed strictly for performance and their failure rates are low. As a result, this type of distributed system is the most efficient. It is also more expensive than the types described below.

Distributed Grids are grids constructed from servers distributed over the Internet. Distributed grids have higher failure rates for jobs, but very low administrative burden for the grid administrator. The Xgrid agent (the server) can be associated with a specific controller by assigning the IP address or host name for its desired controller. This type of computing is the focus of this thesis.

Local Grids are distributed grids where servers are distributed over intranets under the administration of one organization. All the testing performed in this thesis is done on such systems.

2.3.4 Xgrid Components

Figure 2.3 shows how Xgrid works. Every Xgrid system is mainly constructed from three components: agents, clients and a controller.

Agents are the servers that run the computational jobs. Essentially an agent in Xgrid is a Mac OS X computer with the Xgrid service (daemon) turned on. By default this service is turned off. When the Xgrid agent is turned on, it becomes active at startup and it registers itself with a controller. An agent can be registered with only one controller at a time. By default, agents seek to bind to the first available controller on the local network. Alternatively, a specific controller can be specified for an agent.

The agent's controller sends instructions and data to the agent. Upon receiving the data from the controller, the agent starts the job execution and sends the results back to the controller when finished. The agent can be set to accept instructions at any time, however, the default behaviour is to accept tasks only when idle and when the agent has not received any user input for at least 15 minutes.

A **Client** is any Mac OS X machine running Mac OS X v10.4 (Tiger) or later, or Mac OS X Server v10.4, and has a network connection to the Xgrid controller. Job submission is usually done by a command-line tool accessed with the Terminal

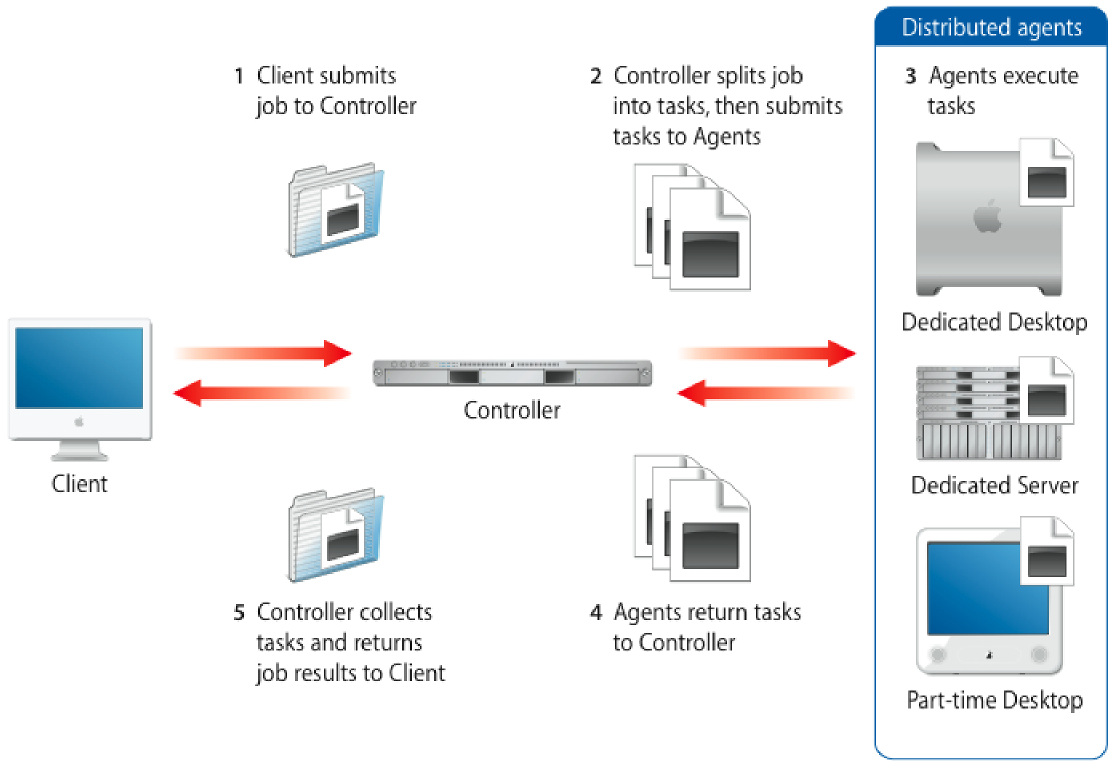

Distributed agents
1 Client submits job to Controller
2 Controller splits job into tasks, then submits tasks to Agents
3 Agents execute tasks
Dedicated Desktop
Dedicated Server
Part-time Desktop
Client
Controller
5 Controller collects tasks and returns job results to Client
4 Agents return tasks to Controller

Figure 2.3: Example of how Xgrid works [8]

Application on Mac OS X. In the case of password protection, the protected controller cannot accept jobs from any Xgrid client unless a valid password is provided with the job submission.

A **Controller** manages the communications among the agents in an Xgrid system. The controller accepts connections from clients and agents. It receives job submissions from the clients, breaks the jobs up into tasks, dispatches tasks to the agents, and returns results to the clients after receiving the results from the agents.

Every logical grid can have one controller. The theoretical maximum number of agents connected to a controller is the number of available sockets on the controller system.

2.3.5 Xgrid Advantages

After investigating several desktop grid technologies, Xgrid technology was chosen to be employed in our system for several reasons:

- The number of departmental machines that can run Mac OS X. The department has more than 50 machines that are running Mac OS X.
- Simple grid configuration and deployment. The process of configuring an Xgrid system is neither complex nor time consuming.
- Straightforward yet flexible job submission. This flexibility can be exploited if the system were to be extended. Adding new nodes is simple.
- Flexible architecture based on open standards.
- Supports command-line interface. This enables testing and enables the building of software components on top of the Xgrid software to be automated.
- Xgrid has a good community around it. The Xgrid community was helpful in the process of development. It also suggests that the Xgrid technology will evolve and enjoy a long life span.
- Stability and reliability. It is used in large scale projects and has been tested extensively by users.
- Password-based authentication support. This enables us to control access to the system without building a security layer.

2.4 Future of Desktop Grids

Grid computing is currently an active research field. Several conferences are held yearly. Grid computing technology is supported by large corporations such as IBM and Apple. In addition, there are open architecture standards (e.g. BOINC) which suggests that the future development will be standardized and open.

Another important factor that will determine the success of this technology in the future is the commercial adoption of this technology. In [33] IBM states that:

> *Over the last few years we have seen grid computing evolve from a niche technology associated with scientific and technical computing, into a business-innovating technology that is driving increased commercial adoption.*

This commercial side of grid computing can be seen through the existence of services like IBM Grid and Grow^{TM}[33] and companies like Platform^{TM}[44], which suggests that grid computing is a big part of the future of super computing.

Chapter 3

Workload and Availability Models

3.1 Workload Model

The theoretical workload model assumed in this thesis is the same model followed by the authors of [3]. Hence the materiel of this section is taken from that resource.

In the assumed model for a Desktop Grid there is a dedicated *Mapper*. This *Mapper* is responsible for scheduling and assigning incoming requests for resources to the available resources in the Desktop Grid. The number of machines in the Grid is M. It is assumed that the jobs are classified into N classes. Jobs of the same class have common characteristics. Let J be the set of machines and I the set of classes. Jobs that belong to the same class i have arrival rate α_i. Let α be the arrival rate vector, then the ith element of α is α_i. Moreover, the average execution rate that a machine j can execute a job from class i is denoted by $\mu_{i,j}$. The actual execution rate is $\mu'_{i,j} = \mu_{i,j}.a_j$ where a_j is the availability (given as a percentage) of the machine (more details are provided in Section 3.2). In addition, μ_i is a vector that represents the execution rates for a particular job class. The j^{th} element in this vector is $\mu_{i,j}$. Finally, μ is the matrix constructed by all execution rate vectors, where the entry (i, j) is $\mu_{i,j}$.

The jobs in the model are assumed to be independent and atomic. They are independent in the sense that the execution or the result of a job does not depend on any other results of other jobs. Also, jobs are atomic in the sense that every job is one complete unit and not a part of a larger job.

Pull-based scheduling is mainly used in resource management systems for Desktop Grids [13]. Pull-based scheduling is a type of scheduling driven by servers announcing their availability in order to be assigned a new job for execution. Please refer to Section 4.2 for more details. In Desktop Grids, using pull-based scheduling is necessary due to the property that the servers are not dedicated.

One of the results of using pull-based scheduling in Desktop Grids is that jobs queue at the Mapper, hence there is no queuing at the servers. In fact in our model, at most one job at a time can be executed without pre-emption on a server [19]. In addition to that, in pull-based scheduling, the scheduler makes a decision as soon as it receives a request from a machine. This is different from on-line mode mapping where a mapping decision is made by the mapper as soon as a job arrives [36].

Servers can fail or become unavailable at any time without any advance notice [7]. When a server fails while executing a job, then that job must be be resubmitted to the mapper. It is assumed that the mapper becomes aware of the failure within a negligible amount of time [32]. Moreover, it is assumed that the Desktop Grid is used to execute short-lived applications [32]. Hence, in such systems, fault tolerant scheduling mechanisms such as checkpointing, migration and replication are not considered, due to their overhead.

3.2 Availability Model

The main difference between cluster-based grids and desktop grids is that for the latter the availability of machines and CPUs changes with time. The machine availability is a binary value that indicates if a machine is reachable. Machines may become unavailable due to communication failure or machine reboot, for example. The CPU availability is a percentage that quantifies the fraction of the CPU time that can be exploited by desktop grid applications [3]. (A brief literature review of availability models and prediction methods can be found in [3].)

In [39], an approach for predicting machine availability in desktop grids is presented. In this approach, a semi-Markov process is applied for prediction. The experiments in [39] suggested that their prediction method has an accuracy of 86%. They also showed the effectiveness of their scheduling policies in large compute-bound guest applications. The policies considered in this thesis assume short-lived applications. In [39] the week was divided into weekdays and weekends and every day was divided into hours.

We assume a strong correlation between the availability of a machine in a particular time and the availability of the same machine in previous weeks around the same period of time (e.g. the availabilities of a machine around noon on successive Mondays are related). A similar assumption is made in [39].

In our availability model we divided the time into days, with each day divided into N equal intervals. The number of intervals and consequently their length is defined by the tester.

The predicted availability in a specific interval i is calculated using the previous readings in the same interval i from the previous weeks. Let $d \in D$, where $D = \{Mo, Tu, We, Th, Fr, Sa, Su\}$ (a day of the week), $a^d_{i,j}$ is the availability for interval i of day d in the j^{th} week, $\hat{a}^d_{i,j}$ is the estimated availability for interval i of day d in the j^{th} week, w_k is the weight given to the the reading $a_{i,j-k}$ and c is a number in [0, 1]. The current implementation has a choice of methods to estimate availability. The first one is

$$\hat{a}^d_{i,j} = \frac{\sum_{k=1}^{N} w_j a^d_{i,j-k}}{\sum_{k=1}^{N} w_k} \tag{3.1}$$

and the second is

$$\hat{a}^d_{i,j} = c a^d_{i,j-1} + (1-c)\hat{a}^d_{i,j-1}. \tag{3.2}$$

Using (3.1) gives the tester flexibility in choosing the weights for previous readings and also the number of previous readings considered. Using (3.2) takes a different approach by considering all previous readings. Also, the tester is given the flexibility of choosing the value of c. This recursive prediction method is a typical way of predicting time related properties or events [50].

As part of the testing environment, an availability prediction module that implements (3.1) and (3.2) was developed. The availabiliy predictor was implemented as the java package *pulling.availability_prediction*.

The default values for (3.1) are $N = 4$ and $w_1 = 0.4, w_2 = 0.3, w_3 = 0.2, w_4 = 0.1$. N was chosen to be 4 to include the effect of the readings from the previous month only. The weights were chosen in a way that gives more weight to recent readings than older ones. As for (3.2), the default values are $c = 0.5$ and $\hat{a}_{0,j} = 0.5$, $\forall j$. Also, (3.1) is the default method.

Chapter 4

Scheduling Schemes

4.1 Introduction

Desktop grid systems allow the development of applications for computationally intensive problems and sustain throughputs far exceeding those of much more expensive supercomputers. To achieve this efficiently, a *scheduling policy* is deployed.

The basic function of a scheduler, which applies the scheduling policy, is to accept requests for resources from clients and assign specific server resources from the pool of grid resources in a specific order, to achieve a certain goal. In other words, scheduling is the process of assigning requests or tasks to the most suitable resource provider (i.e. where to execute a task) and ordering requests or tasks (i.e. when to execute tasks) [14]. Each scheduling policy is designed to optimize certain performance requirements. Also, different scheduling polices require different information on the system state (e.g. arrival rates and machine execution rates).

A scheduling policy must be scalable, i.e. applicable to large-scale systems involving large numbers of computers. This scalability requirement may increase the complexity of scheduling policies. The complexity is further complicated by several factors. First, the scheduling policy must be fault-aware and cope with resource volatility since resources can be disconnected from the grid at any time without any advance notice. For example, a volunteer computer may be restarted or the resources of a connected computer may become fully occupied with local jobs. Furthermore, as desktop grids are constructed using volunteer computers, the resources are not fully dedicated. Thus, a scheduling policy must exploit the available knowledge of the effective computing power contributed by resources, which also adds additional complexity [3].

Another contributing factor is related to the heterogeneous nature of desktop grids. Scheduling polices that do not consider information on jobs and machine heterogeneity will perform poorly in heterogeneous environments. There is already work on developing polices for cluster systems of dedicated and heterogeneous machines (see Al-Azzoni and Down [4], He *at al.* [36] and Maheswaren *et al.* [43]). As for heterogeneous desktop grids the authors of [3] state that their paper "Dynamic Scheduling for Heterogeneous Desktop Grids" is the first paper to consider the problem of scheduling for heterogeneous Desktop Grids involving resource volatility.

The Linear Programming Based Affinity Scheduling policy for desktop grids (LPAS_DG) policy introduced in [3] will be discussed later in this chapter after introducing a taxonomy of Desktop grids and scheduling polices.

4.2 Taxonomy of Desktop Grids

In [14], a taxonomy of desktop grid systems is suggested. This section is a summary of that work. Desktop grids are categorized according to organization, platform, scale and resource provider properties. Please refer to Figure 4.1.

4.2.1 Organization

In terms of organization, desktop grids can be divided into two categories according to the organization of components.

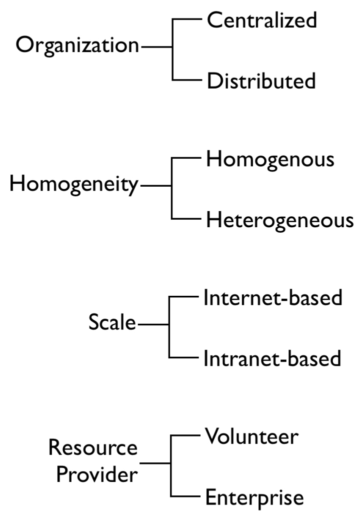


Figure 4.1: Taxonomy of Desktop Grids.

Centralized Desktop Grids

A centralized desktop grid consists of clients, resource providers (or volunteer servers), and a mapper (scheduler). The execution model of centralized desktop grids consists of the following phases [14]:

- *Registration phase*: Resource providers register their information to the mapper.
- *Job submission phase*: A client submits a job to the mapper.
- *Resource grouping phase*: A mapper constructs a Computational Overlay Network (CON) according to capability, availability, reputation, trust of resource providers, etc. A CON is a set of resource providers. Scheduling is conducted on the basis of a specific structure or topology. This step depends on the assumed workload model.
- *Job allocation phase*: The mapper assigns tasks to servers.
- *Job execution phase*: Servers execute their tasks.

- *Task result return phase*: Servers return results to the mapper.
- *Task result certification phase*: The mapper checks the correctness of the returned results. This step is done if the scheduling policy deploys a verification mechanism.

Distributed Desktop Grids

A distributed desktop grid consists of clients and resource providers (or volunteer servers). Distributed desktop grids lack the existence of a centralized mapper. However, volunteering servers have partial information about other volunteers. Volunteers are responsible for constructing CONs and scheduling a job in a distributed fashion. The execution model of distributed desktop grids is as follows:

- *Registration phase*: Servers exchange their information with each other.
- *Job submission phase*: A client submits a job to a neighbouring server.
- *Resource grouping phase*: Servers self-organize CONs according to several factors (e.g. capability, availability and reputation).
- *Job allocation phase*: A server assigns the job to other servers either to execute or to schedule inside their CON according to a distributed scheduling policy.
- *Job execution phase*: Servers execute their task.
- *Task result return phase*: Servers return results to their parent servers.
- *Task result certification phase*: The parent server checks the correctness of the returned results. This step is executed if the scheduling policy deploys a verification mechanism.

4.2.2 Homogeneity

Desktop grids can be homogenous or heterogeneous. In **homogenous grids**, the execution rates of two servers for different task classes are proportional; for instance, if server s_1 is twice as fast as server s_2 in executing tasks from class 1, then server s_1 will be twice as fast as server s_2 in executing all other classes. In **heterogeneous grids** execution rates of different servers are not correlated. For example, in a heterogeneous grid one might find a server s_1, which is much faster than server s_2 in executing tasks that require a lot of matrix computations, while being slower on other task classes that require different kinds of computations.

4.2.3 Scale

Desktop grids are categorized into Internet-based and Intranet-based. **Internet-based** Desktop grids are constructed from servers around the Internet. On the other hand, **intranet-based** desktop grids are based on servers within a corporation, a university, or an institution. This type has more availability than Internet-based desktop grids, however it is usually much smaller [14].

4.2.4 Resource Provider

Desktop grids are categorized into volunteer and enterprise categories. **Volunteer** grids are constructed from servers whose owners willingly donate the idle time of their machines. This type of grid is normally Internet-based. One the other hand, **enterprise grids** are grids consisting of servers owned by a single organization, and this type is usually Intranet-based [14].

4.3 Taxonomy of Desktop Grid Scheduling Policies

In [14], a taxonomy of desktop grid systems from the perspective of the scheduler (mapper) is suggested. This section is a summary of a taxonomy of mappers derived from that work. Mappers can be categorized according to organization, mode, scheduling policy complexity, dynamism, adaptation and fault tolerant approaches (Figure 4.2). The remainder of this section is an elaboration of each category.

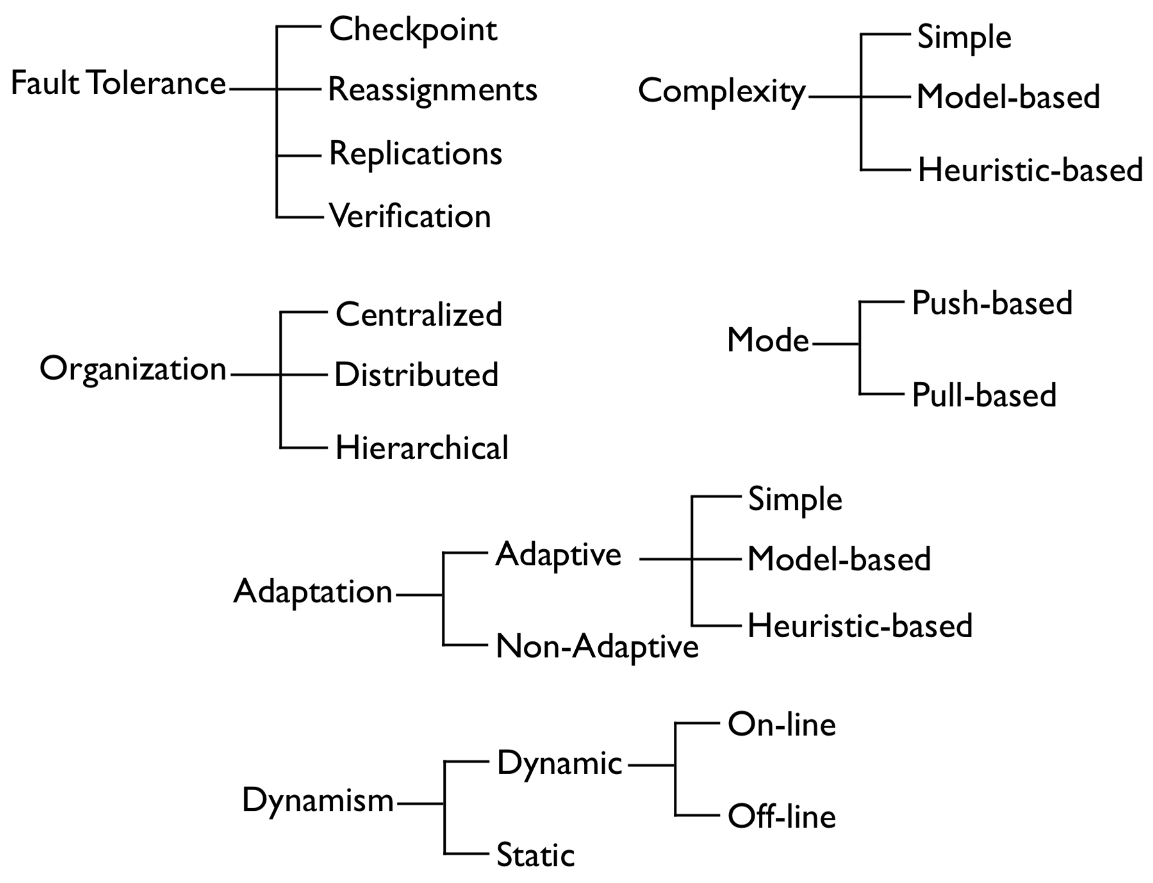


Figure 4.2: Taxonomy of Desktop Grid Mappers.

4.3.1 Organization

The organization of a scheduling scheme can be classified into three categories: centralized, distributed, and hierarchical, depending on where and how scheduling decisions are made. In the **centralized approach**, there is a central scheduler responsible for the scheduling process. The central scheduler maintains all grid status information. On the contrary, in the **distributed approach**, scheduling decisions are the joint responsibility of all of the servers in the system. Each server has some partial information about the system status that it uses in making its scheduling decisions. Finally, in the **hierarchical approach**, scheduling decisions are performed in a hierarchical fashion, where high level schedulers perform scheduling and assign tasks to low level schedulers, which perform scheduling in a centralized way for their group of servers [14].

4.3.2 Mode

Depending on when the scheduling process is initiated. Scheduling policies can be categorized into two modes [14].

Push-based Mode

In this mode, the scheduling process is initiated when a task is submitted, and ends when the scheduler assigns (or pushes) the task to a certain server according to the scheduling policy. This mode is not common in desktop grids, due to their dynamic nature and the fact that servers are not dedicated.

Pull-based Mode

In this mode, the scheduling process is initiated when a server declares its availability, in other words when a server requests (or pulls) tasks from its mapper. This mode is more common in desktop grids, due to their dynamic nature and the fact that servers are not dedicated.

4.3.3 Scheduling Policy Complexity

In terms of complexity, schedulers can be divided into into three categories [14]: simple, model-based, and heuristics-based.

In the **simple approach**, tasks and resources are selected by using a simple approach like First Come First Served (FCFS) or the random scheduling policy [14].

The **model-based approach** is divided into deterministic, economy, and probabilistic models. The *deterministic model* is based on a data structure or topology such as queue, stack, tree, or ring. Tasks or resources are deterministically mapped according to the properties of structures or topologies. For example, in a tree topology, tasks are allocated from parent nodes to child nodes. In the *economy model*, scheduling decisions are based on financial factors where priorities are given to tasks according to the price paid by the job submitter. In the *probabilistic model*, resources are selected using probabilistic models (e.g. Markov processes or genetic algorithms) [14].

In **the heuristics-based** approach, tasks and resources are selected by using ranking, matching and exclusion methods based on performance, capability, weight, precedence, workload, availability, location, reputation/trust, etc. The *ranking method* ranks the resources (servers) and tasks according to quantifying criteria and then selects the most suitable resource and task (e.g. the largest or the smallest one). The *matching method* selects the most suitable tasks and resources in accordance to evaluation criteria (e.g. min-min, max-min, sufferage, etc.). The *exclusion method* excludes resources according to a specific criterion, and then chooses the most appropriate one among the remaining set of resources. Ranking, matching, and exclusion methods can be used together or separately. Criteria used in these methods are numerous and the following are some of them: arrival time and task class are used only for tasks while availability, performance, capability, location, and reputation are used for servers [14].

4.3.4 Dynamism

Scheduling schemes are categorized as static or dynamic depending on the information taken into consideration when making the scheduling decision. In the case of **static scheduling**, the state information that the policy is aware of does not change with time, hence no dynamic information about the resources (servers) is taken into account (e.g. availability) when scheduling. Such a policy use the static information from prior knowledge (prior to the start of the scheduling process) combined with the request status to come to a scheduling decision. For example, the First Come First Served policy is a static scheduling policy where only the prior static knowledge about the servers and the dynamic information about requests are used. On the other hand **dynamic scheduling** takes the general system status into consideration. Availability information, performance information and the absence or presence of servers are examples of the changing information that dynamic policies take into account when making a scheduling decision. Dynamic scheduling policies cope well with the fact that some servers may go off-line and others may join the grid, thus they suit large scale Desktop grids.

Dynamic scheduling is further classified into online and periodic depending on the time at which the scheduling event occurs. In *the online approach*, scheduling is started by the arrival of a new task or a resource provider. In the *periodic approach*, scheduling events occur periodically at a predefined interval.

4.3.5 Adaptation

Based on adaptation, scheduling schemes can be categorized as being adaptive or non-adaptive.

Adaptive scheduling

Adaptive scheduling takes environmental stimuli into account to adapt to dynamically changing environments. The environmental change leads to a change in the scheduling policy to obtain better performance. There are several types of adaptive scheduling mechanisms, and the following is list of them:

Migration

Migration is where tasks are moved from one server for some reason (for example, moving a task from a server which has become busy with local jobs to a less busy server).

Redundant assignment

Redundant assignment allows the assignment of the same task to more than one server. Some policies may always allow redundant assignment to achieve replication, others may only allow it under defined conditions (e.g. when the first assigned server times out).

Change Policy

In this approach the scheduling policy used can be switched in order to cope with new conditions dictated by environmental change. For example the Minimum Completion Time policy can be switched to the Minimum Execution Time policy when the system load distribution changes.

Non-adaptive scheduling

Non-adaptive scheduling does not take environmental stimuli into account.

4.3.6 Fault Tolerant

Scheduling policies are different in the way that they deal with faults. There are several approaches that may be taken.

Checkpoints

In this approach, the current state of an active task is saved at different points through the execution process in a manner such that if a failure was detected the failed task can be restarted from the most recently saved point (called a checkpoint) on a different server.

Reassignment

When a scheduler detects a failure in a server, it simply reassigns the task to another server.

Replication

In this approach, replication is used as a method of fault tolerance. In case a server failed when executing a specific task, the result can still be obtained from another server executing the replicated task. This method anticipates failures, whereas Reassignment methods reacts to failures.

Result Certification

Scheduling polices can have a mechanism to validate results (or part of them) to guarantee their correctness.

4.4 Scheduling Policies

In the following subsections different scheduling policies will be discussed. Some of these scheduling schemes will be used in the experiments discussed in Chapter 7.

4.4.1 FCFS

First Come First Serve (FCFS) is one of the most basic scheduling policies. When a server is available, the job that has been waiting the longest is assigned to that server regardless of the processing rate of the server. The FCFS policy is easy to implement and it does not add an overhead to the scheduling process since it does not maintain a large amount of data, neither does it perform expensive calculations. However, we will see that this scheme can perform poorly in heterogeneous environments (see the experiments in Chapter 7).

4.4.2 MET

Minimum Execution Time (MET) is a static scheduling policy. A mapper using the MET policy always gives the fastest machines the highest priority. An incoming task is assigned to the machine that has the least expected execution time for the task. Thus, when a new task of class i arrives in the system, the mapper assigns it to machine $j \in arg\ min_{j'} 1/\mu_{i,j'}$ [36]. As defined in Section 3.1, $\mu_{i,j'}$ is the processing rate of machine j' for class i. Ties are broken arbitrarily; for example, the mapper could pick the machine with the largest index j when more than one machine has the minimum expected execution time [4].

This heuristic enjoys an advantage of not requiring machines to send their expected completion times back to the mapper as tasks arrive, neither does it require them to send availability (i.e. use effective processing rates), thus the MET policy requires limited communication between the mapper and machines.

However, it may suffer from severe load imbalance, even causing the system to become unstable. An illustration for such a case is a system with two machines and one stream of tasks with rate $\alpha_1 = 6$, and the execution rates are $\mu_{1,1} = 5$ and $\mu_{1,2} = 3$ for machine 1 and machine 2 respectfully. When the MET heuristic is used, all tasks are mapped to machine 1, since the execution rate of machine 1 is larger than that of machine 2. In this case the system will be unstable because tasks are arriving to the system at a rate larger than that at which they are served ($\alpha_1 > \mu_{1,1}$). It is easy to see that this instability can be avoided if an adequate portion of tasks were assigned to machine 2, since $\alpha_1 < (\mu_{1,1} + \mu_{1,2})$. The information used by this policy is known prior to the start time of the mapping, making it a static policy.

4.4.3 MCT

Minimum Completion Time (MCT) is a dynamic scheduling policy. A mapper using MCT assigns an arriving task to a machine that is expected to complete the task the earliest, hence the term minimum completion time [36]. Minimum completion time is calculated from two terms. The first includes the execution rates of the machines for the arriving task class, and the second is how long machines are expected to be busy for executing current tasks.

The MCT policy is stated formally as follows. When a task of class i arrives the mapper assigns it to a machine j such that

$$j \in arg\ min_{j'}\{1/\mu_{i,j'} + \Sigma_{i' \in I} Q_{i',j'}/\mu_{i',j'}\} \tag{4.1}$$

where $Q_{i',j'}$ is the number of tasks of class i' that are executing or waiting at machine j', at the time of the arrival of task i. The mapper examines all the machines in the grid system to find out the machine with the earliest expected completion time [4].

One drawback of this heuristic is that the mapper requires machines to send their expected completion times, which might result in communication overhead in the grid.

However, MCT mitigates the load imbalance that happens when using MET. To illustrate how load imbalance is avoided, let us look again at the example from the previous section. As a reminder, the system has two machines and one stream of tasks arriving at rate $\alpha_1 = 6$, and the execution rates are $\mu_{1,1} = 5$ and $\mu_{1,2} = 3$ for machine 1 and machine 2 respectively. Under the MCT policy, when the first task arrives (let this task be of class 1), the mapper assigns it to machine 1 since its expected completion time is earlier than that of machine 2 (see (4.1)). If a second task arrives within k time units where $k < (1/\mu_{1,1} + Q_{1,1}/\mu_{1,1} - 1/\mu_{i,2} + Q_{1,2}/\mu_{1,2})$, the mapper will assign it to machine 2 despite the fact that machine 1 is faster than machine 2 in executing the task. This is because machine 1 will be busy executing the first task and the completion time for the task is less if sent to machine 2 (see (4.1)). The fact that the mapper considers how busy the machines are, results in mitigating the load imbalance problem from which MET can suffer.

Several existing resource management systems use the MCT policy or other polices that are based on the MCT policy, including SmartNet [23].

4.4.4 KPB

The k-Percent Best (KPB) policy attempts to combine advantages of both the MET and the MCT policies [36]. Upon the arrival of a task the mapper chooses the $(kM/100)$ best machines based on their execution times for the task class, where $100/M \leq k \leq 100$. Then, the mapper assigns the task to the machine with the earliest expected completion time among the machines in that subset [4]. This policy first uses MET on all the machines in order to pick the $(kM/100)$ best machines and then uses MCT on that subset of machines to pick a machine to send the task to. Doing this not only guarantees that the task will be sent to a superior machine in terms of execution rate (a guarantee that MET can offer), but also takes current machine loads into consideration (a property of MCT).

The KPB policy needs only to communicate with the subset of machines first, rather than with all of the machines in the grid. Another advantage for this policy is that it attempts to avoid assigning the task to a machine that could do better for tasks that arrive later [4].

The optimal value of k varies depending on the number of machines, execution rates and arrival rates. The KPB policy can perform poorly relative to the MCT policy if some machines are not among the best $k\%$ for any task class [4]. Also, if $k = M$, then the KPB policy is identical to MCT. On the contrary, if $k = 1$, then the KPB policy is identical to MET.

4.4.5 Gcμ

This scheduling policy is a variation of the generalized cμ (Gcμ) policy [34]. This policy asymptotically minimizes delay costs. When a machine j requests a task, the scheduler assigns it the longest waiting job from class i such that $i \in arg\, max_i D_i(t)\mu'_{i,j}$ [3]. The use of this policy in desktop grids was first suggested in [3]. The optimality of this policy is obtained under heavy loads (i.e. loads that approach 100%). On the other hand under more moderate load, this policy can make bad scheduling decisions especially with heterogeneous machines. This happens because the policy assigns an arriving job to the fastest machine available without considering the execution rate of this machine for different job classes. For example let us assume the following system:

$$\alpha = \begin{bmatrix} 1 & 1.5 \end{bmatrix} \text{and } \mu = \begin{bmatrix} 2 & 2 \\ 2.1 & 10 \end{bmatrix}.$$

If machine 2 becomes available and there are two jobs from each class, the scheduler will assign to it the job from Class 1. The greedy nature of this policy prevents it from choosing a job from Class 2 which machine 2 can execute quickly.

Nonetheless, the G$c\mu$ policy results in achieving significant performance improvement over simpler scheduling schemes such as FCFS. This improvement is a result of using the execution rates when making scheduling decisions that attempt to assign jobs to machines which will execute these jobs faster than any of the other available jobs.

4.4.6 LPAS_DG

The <u>L</u>inear <u>P</u>rogramming Based <u>A</u>ffinity <u>S</u>cheduling policy for <u>D</u>esktop <u>G</u>rids (LPAS_DG) was proposed in [3]. The description here is exactly as in the original publication.

> "The <u>L</u>inear <u>P</u>rogramming Based <u>A</u>ffinity <u>S</u>cheduling policy for <u>D</u>esktop <u>G</u>rids (LPAS_DG) requires solving the following allocation Linear Problem (Andradóttir *et al.* [5]) at each machine availability/unavailability event, where the decision variables are λ and $\delta_{i,j}$ for $i = 1, \ldots, N$, $j = 1, \ldots, M$. The variables $\delta_{i,j}$ are to be interpreted as the proportional allocation of machine j to class i.

$$\begin{aligned} \max\ & \lambda \\ \text{s.t.}\ & \sum_{j=1}^{M} \delta_{i,j}\mu'_{i,j} \geq \lambda\alpha_i, \qquad \text{for all } i = 1,\ldots,N, && (4.2) \\ & \sum_{i=1}^{N} \delta_{i,j} \leq 1, \qquad \text{for all } j = 1,\ldots,M, && (4.3) \\ & \delta_{i,j} \geq 0, \qquad \text{for all } i = 1,\ldots,N, \text{ and } j = 1,\ldots,M. && (4.4) \end{aligned}$$

The left-hand side of (4.2)[1] represents the total execution capacity assigned to class i by all machines in the system. The right-hand side represents the arrival rate of tasks that belong to class i scaled by a factor of λ. Thus, (4.2) enforces that the total capacity allocated for a class should be at least as large as the scaled arrival rate for that class. The constraint (4.3)[2] prevents overallocating a machine and (4.4) states that negative allocations are not allowed.

Let λ^* and $\{\delta^*_{i,j}\}$, $i = 1,\ldots,N$, $j = 1,\ldots,M$, be an optimal solution to the allocation LP. The allocation LP always has a solution, since no lower bound constraint is put on λ. Let δ^* be the machine allocation matrix where the (i,j) entry is $\delta^*_{i,j}$.

Whenever a machine becomes available or unavailable, the scheduler solves the allocation LP to find $\{\delta^*_{i,j}\}$, $i = 1,\ldots,N$, $j = 1,\ldots,M$. If a machine j becomes unavailable, then $a_j = 0$. In this case, $\delta^*_{i,j} = 0$ for $i = 1,\ldots,N$. On the other hand, if a machine j becomes available, a_j is equal to the predicted CPU availability for machine j during its next expected machine availability period. The scheduler obtains values for a_j using the CPU availability prediction techniques discussed in (the previous section)[3]. Solving the allocation LP at each availability/non-availability event represents how the LPAS_DG policy adapts to the dynamics of machine availability. Constraint (4.3) enforces the condition that the allocation of machine j should not exceed its CPU availability. The use of a_j represents how the LPAS_DG policy adapts to the dynamics of CPU availability.

The value λ^* can be interpreted as follows. Consider an event in which a machine becomes available or unavailable. Let λ^* and $\{\delta^*_{i,j}\}$, $i = 1,\ldots,N$, $j = 1,\ldots,M$, be an optimal solution to the allocation LP corresponding to the system just after the occurrence of the event. Consider the system that only consists of the available subset

[1]$\mu'_{i,j}$ is defined in Section 3.2.
[2]a_j is defined in Section 3.2.
[3]Previous section in original paper. Please refer to Section 3.2 in this thesis.

of the M machines. Then, the value λ^* can also be interpreted as the maximum capacity of this partial system [4, 26].

The LPAS_DG policy is defined as follows. When a machine j requests a task, let S_j denote the set of task classes i such that $\delta^*_{i,j}$ is not zero ($S_j = \{i : \delta^*_{i,j} \neq 0\}$). Let $D_i(t)$ be the waiting time (sojourn time) of the head of the line class i task at the time t of making the scheduling decision. The scheduler assigns machine j the longest-waiting (head of the line) class i task such that

$$\mu_{i,j}\delta^*_{i,j} > 0 \text{ and } i \in \arg\max_i \mu_{i,j} D_i(t).$$

Note that $\mu_{i,j}$ represents the effective execution rate for class i tasks at machine j ($\mu_{i,j} = a_j \mu'_{i,j}$ for $i = 1, \ldots, N$, $j = 1, \ldots, M$). Note that the LPAS_DG policy does not use the actual values for $\{\delta^*_{i,j}\}$, beyond differentiating between the zero and nonzero elements. Regardless, we must solve the allocation LP to know where the zeros are.

The allocation LP considers both the arrival rates and execution rates and their relative values in deciding the allocation of machines to tasks. In addition, these allocations are constrained by the CPU availabilities of the available machines. Consider a system with two machines and two classes of tasks ($M = 2$, $N = 2$). The arrival and execution rates are as follows:

$$\alpha = \begin{bmatrix} 1 & 1.5 \end{bmatrix} \text{ and } \mu = \begin{bmatrix} 9 & 5 \\ 2 & 1 \end{bmatrix}.$$

Assume that all machines are dedicated (*i.e.,* $a_j = 1$, for all $j = 1, \ldots, M$). Solving the allocation LP gives $\lambda^* = 1.5789$ and

$$\delta^* = \begin{bmatrix} 0 & 0.6316 \\ 1 & 0.3684 \end{bmatrix}.$$

Thus, when machine 1 requests a task, the scheduler only assigns it a class 2 task. Machine 2 can be assigned tasks belonging to any class. Although the fastest rate is for machine 1 at class 1, machine 1 is never assigned a class 1 task. Note that machine 1 is twice as fast as machine 2 on class 2 tasks and note that $\frac{\mu_{1,1}}{\mu_{2,1}} < \frac{\mu_{1,2}}{\mu_{2,2}}$.

Now assume that machine 1 is fully dedicated and machine 2 is half-dedicated (*i.e.,* $a_1 = 1$ and $a_2 = 0.5$). Solving the new allocation LP gives $\lambda^* = 1.3143$ and

$$\delta^* = \begin{bmatrix} 0.0143 & 0.5 \\ 0.9857 & 0 \end{bmatrix}.$$

In this case, machine 1 is assigned tasks from any class, but machine 2 is only assigned class 1 tasks. Note that machine 1 is four times as fast as machine 2 on class 2 and thus the LPAS_DG policy avoids assigning a class 2 task to machine 2.

There could be many optimal solutions to the allocation LP. These optimal solutions may have different numbers of zero elements in the δ^* matrix. The following proposition is a basic result in linear programming (the proof can be found in Andradóttir *et al.* [6]):

> *There exists an optimal solution to the allocation LP with at least $NM + 1 - N - M$ elements in the δ^* matrix equal to zero.*

Ideally, the number of zero elements in the δ^* matrix should be $NM + 1 - N - M$. If the number of zero elements is greater, the LPAS_DG policy would be significantly restricted in shifting workload between machines resulting in performance degradation. Also, if the number of zero elements is very small, the LPAS_DG policy resembles more closely the Gcμ policy. In fact, if the δ^* matrix contains no zeros at all, then the LPAS_DG policy reduces to the Gcμ policy.

The LPAS_DG policy can be considered as an adaptive policy. As the policy only involves solving an LP, it is suited for scenarios when the global state of the system changes. For example, new machines can be added and/or removed from the system. Also, parameters such as the arrival rates and execution rates may change over time. On each of these events, one needs to simply solve a new LP and continue with the new values."

Chapter 5

System Design

The system considered in this thesis is purely a software system. New software components were designed, implemented and tested and thereafter combined with existing software to create the desired testing environment.

5.1 System Requirements Specification

This section discusses the requirements for the scheduling schemes testing environment. The requirements imply a set of attributes that the final product must achieve. The requirements are used throughout the software life cycle.

5.1.1 Purpose

The purpose of this software system is to create a testing environment for scheduling schemes. This environment should allow testers to program new scheduling schemes and then test them.

5.1.2 User Classes and Characteristics

The expected users of this system are researchers in the scheduling field. The users will use the system by testing built in scheduling schemes or adding new ones.

5.1.3 User Documentation

In addition to the software system, user documentation is provided. Four types of documentation were prepared.

1. **User manual** explaining the functions of the software and how the software is used to achieve these functions.

2. **Expansion Documentation** explaining how a tester can expand the system by adding new features and layers to the system such as scheduling schemes or probability distributions.

3. **Javadoc Documentation** that explains the classes, their attributes and their methods.

4. **Code Documentation** explaining how the code works and why. This along with the Javadoc documentation should help in future modifications.

5.1.4 Functional Requirements

A system's functional requirements define its behaviour. The following is a list of the functional requirements for the system implemented:

1. The system shall use the workload model defined in Section 3.1.

2. The system shall allow the addition of a new scheduling scheme by adding a single Java class.

3. The system shall allow the addition of a new probability distribution by adding a single Java class.

4. The system shall calculate the average waiting time (either overall or by job class) [1].

[1]The waiting time is the difference between the time that a job is submitted and is sent to a server.

5. The system shall calculate the average communication delay (either overall or by job class) [2].

6. The system shall calculate the average response time (either overall or by job class) [3].

7. The system shall allow the testers to define job classes.

8. The system shall contain an availability predictor module.

9. The system shall allow testers to impose heterogeneity on the servers.

10. The system shall allow testers to generate simulated failure traces according to particular probability distributions.

11. The system shall allow the testers to monitor the system activities while in operation (e.g what jobs are currently being executed or what jobs are timed-out, etc.).

12. The system shall be able to generate the same set of jobs from previous tests with the same probability distributions.

13. The system shall log all scheduling events for further study.

14. The system shall log all generations of jobs events for further study.

15. The system shall log all artificial failure events for further study.

16. The system shall log all actual exceptions for further study (e.g communication errors and file exceptions).

[2]The communication delay is the difference between the time a job is sent to be executed and the time the job begins execution. This delay happens mainly due to communication, but it could also be caused by the software layer responsible for the process of distribution and execution of tasks.

[3]The response time is the difference between the time when a job is submitted and when a job completes execution.

5.1.5 Platform Requirements

The system shall run on Mac OS X 10.4 and Mac OS X 10.5.

5.1.6 Maintainability Requirement

The system shall be implemented in a manner allowing for maintenance as well as expansion.

5.1.7 Usability Requirements

The system shall have a Graphical User Interface (GUI) that allows the tester to set up tests and monitor them. In addition, the system shall produce log files that can be opened with spread sheet programs (e.g. Numbers or Excel).

5.2 Design

In this section the software design of the system is explained abstractly from a functional point of view. After analysing the workload model and the requirements of the system, the functions of the software were grouped into six main modules (see Figure 5.1). These modules are discussed in the following subsections.

5.2.1 Classes

The main classes are:

1. *Job Generator*: This module is responsible for generating jobs. As mentioned in Section 3.1, there are N classes of jobs during a specific test. Every class has a different arrival rate. Based on the arrival rate and the underlying inter-arrival time distribution (e.g exponential) chosen by the tester, this module generates jobs for the *Mapper*, simulating submission of jobs to the system. At the implementation level, the *Job Generator* module was implemented as a group of threads and a thread controller. Every job class has a thread which

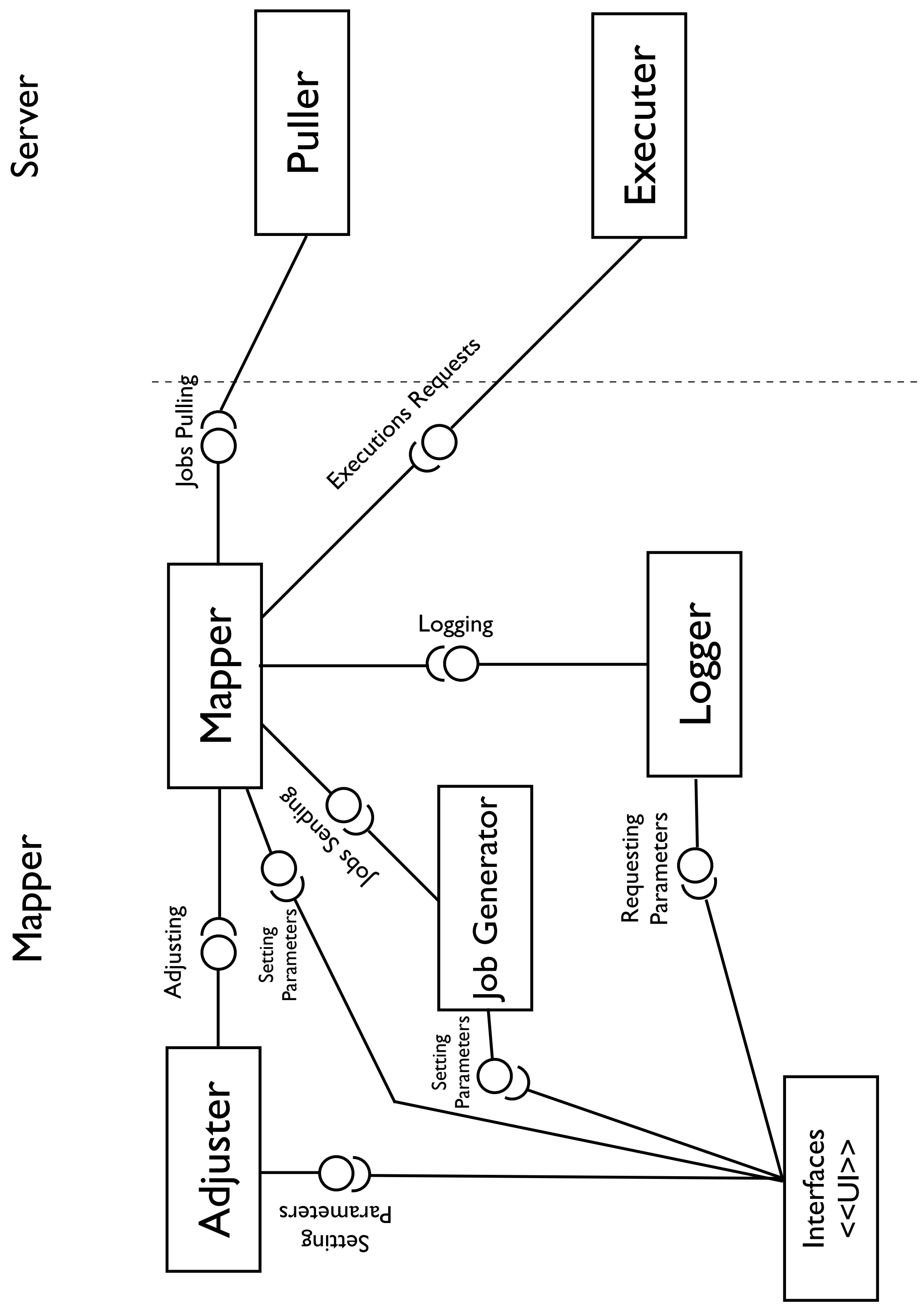


Figure 5.1: Component Diagram of the system

knows the arrival rate and the inter-arrival time distribution of that class and acts accordingly to submit jobs. The module is synthetic in the sense that its job is to simulate users of the system. The *Job Generator* module was implemented as a Java package named *generating.*

2. *Adjuster*: The purpose of this module is to impose some artificial properties on some servers. This module is responsible for adjusting sent jobs to make some servers slower or faster in executing them. For example, if server s can actually execute a job from class A in n time units on average and the tester wishes to increase the execution time for class A to be n' units on average, where $n' > n$, then she should configure the *Adjuster* and set the execution rate of that server to ($1/n'$). The *Adjuster* will then act accordingly, forcing the average execution time to be n' and not n for that specific server. This is used by testers to configure homogeneous systems to be heterogeneous. This module is also used to simulate failures events. This feature might be used to measure the robustness of scheduling schemes when machines fail a certain percentage of time. The module is artificial in the sense that it is used to impose an artificial effect and has nothing to do with the mapping functionality. The *Adjuster* module was implemented as a Java package named *adjusting.*

3. *User Interface*: Its functionality is receiving input from the tester and changing the parameters of the system according to the input. In addition to that, it shows the results of the tests. Thus, this module is responsible for configuring the test at the initial stage and showing the results at the final stage. The module is artificial in the sense that it has nothing to do with the mapping functionality. The Interface module was implemented as the Java package *interfacing.*

4. *Mapper*: As its name indicates this module does the actual mapping. It receives submission requests from the *Job Generator* module and sends each task to a specific server. The process of choosing the server depends on the mapping scheme deployed in that test. The mapping scheme typically needs to

know details about the state of the system (e.g the availability of the servers), therefore, the *Mapper* module keeps track of all of this information. The state information required depends on the mapping scheme itself. Because it is the core of the system functionality, this module is the most complex in the system. In addition to that, it is the module that will be extended by adding additional scheduling schemes. It was designed and implemented to allow this extension. An abstract base-class named *MappingScheme* was defined. This class has a defined and unimplemented set of services (abstract methods). When a new mapping scheme is to be added, a sub-class of the base-class *MappingScheme* should be created. This new sub-class must implement the abstract services. The way these services are implemented determines the new scheduling policy. Please refer to Section 6.4 for details. The *Mapper* module was implemented as the Java package *mapping*.

5. Logger: The functionality of the *Logger* is simple. It keeps a record of the events that happen during the course of a test for further study. This module was implemented as the Java package *logging*.

6. *Puller*: Unlike all of the other modules, this module is deployed at the servers. Every server in the system should have this module running. The *Puller* is responsible for maintaining availability information and sending it to the central *Mapper* notifying it when servers are available. In addition, the *Puller* notifies the *Mapper* when a job is completed. The frequency that the *Puller* monitors the availability is defined by the *Mapper*. We call it system resolution time; it is the time in minutes between two availability readings by the *Puller*. This module was implemented as the Java package *pulling*.

7. *Executer*: This module is responsible for executing and managing the tasks on assigned machines. This module was implemented as the Java package *executing*.

5.2.2 Module Interaction

In the previous subsection the modules in the system were discussed. These modules need to communicate in order for the system to function. In this section the messages between the modules are discussed. The interactions between modules are shown in Figure 5.2.

The *Interface* module sends messages to both the *Adjuster* and the *Job Generator*. The messages sent to the *Adjuster* are used to set the execution rates of jobs classes. The messages sent to the *Job Generator* are used to define the job classes and their arrival rates. Based on these parameters (job classes and their arrival rates) the *Job Generator* module sends messages to the *Mapper* module to submit tasks.

The *Mapper* module sends messages to the *Adjuster*, *Logger* and the *Puller*. The *Adjuster* module is consulted by the *Mapper* before issuing a job to a server, to see what is the execution time expected for that job on that server. In addition to that the *Mapper* informs the *Logger* of every action, so the *Logger* can keep a record of the events happening in the system. Moreover the *Mapper* sends the jobs to the *Executer* module to be executed.

The *Puller* module is responsible for sending availability information to the *Mapper*. It also notifies the *Mapper* when a machine is ready to receive tasks.

The *Executer* module notifies the mapper when a job is done.

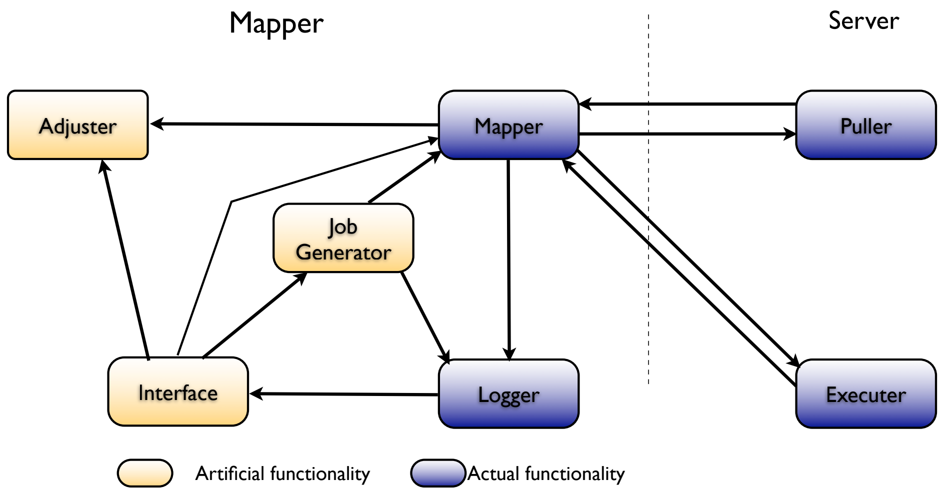


Figure 5.2: Messages between modules.

Chapter 6

System Implementation

This section discusses the implementation phase of the software life cycle.

6.1 Introduction

The software system was implemented in the Java programming language, which was chosen for several reasons. The main reasons were that it is platform independent so testers can use it on any platform they desire, and it is a relatively popular programming language. In addition, Java fits in the object oriented paradigm used in the development of the system. The Eclipse development framework was used for implementation.

After the initial design (Section 5.2), several iterations of refinements took place. After every refinement, a lower level model (in terms of abstraction) was produced. When the last level was reached, each abstract class defined in Figure 5.1 ended up being implemented as a Java package.

The following section (Section 6.2) discusses some related Java topics. Section 6.3 discusses the packages of the software. Following that, Section 6.4 explains how to add new mapping schemes and Section 6.5 explains how to add a new probability distribution. Finally, Section 6.6 touches upon issues considered in the design and development phases.

6.2 Java Related Background

In this section the Java delegation event model and the abstract classes concept are discussed.

6.2.1 Java Delegation Event Model

In [47] the Java delegation event model is described as follows:

> Event types are encapsulated in a class hierarchy rooted at java.util.EventObject. An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type generated. A Listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener. An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface.
>
> An Event Source is an object which originates or fires events. The source defines the set of events it emits by providing a set of set<EventType>Listener (for single-cast) and/or add<EventType>Listener (for multi-cast) methods which are used to register specific listeners for those events.
>
> In an AWT [Abstract Window Toolkit] program, the event source is typically a GUI component and the listener is commonly an "adapter" object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events. The listener object could also be another AWT component which implements one or more listener interfaces for the purpose of hooking GUI objects up to each other.

6.2.2 Abstract Classes

An abstract class in Java is a class that contains an abstract method. Abstract methods are method signatures without implementations. The implementation is provided by the subclasses. Any class that contains abstract methods must be declared abstract. A concrete class is a class without any abstract methods (i.e. all of its methods are implemented). Abstract classes are used to represent abstract concepts that can have several specific instances (concrete instances). For example a mapping scheme is an abstract concept, where the MET mapping scheme is a specific instance of a mapping scheme.

6.2.3 Polymorphism and Dynamic Binding

An object of a subclass in Java can be used by any method written to work with an object of its superclass. This feature is called *polymorphism*. Dynamic binding is binding instances of objects of subclasses to objects of their superclass. For example
Fruit f = new Apple();
f.eat();
when executing *f.eat()* the *eat()* method in the Fruit class is executed.

6.3 Packages

The source code of the system is constructed from eight packages resembling the seven modules of the first abstract design and one helper package. These packages are *adjusting*, *executing*, *generating*, *interfacing*, *logging*, *mapping*, *pulling* and the helper package *probabilityDist*. These packages will be discussed in the remainder of this section.

6.3.1 adjusting

This package is constructed from a single class. This class is called *Adjuster* and contains three methods. The first method is invoked by the active mapper to determine the adjustments to be done on jobs before sending them. The *Adjuster* uses information submitted by the user to perform these adjustments.

6.3.2 executing

The executing package has two classes. The abstract class *Executer* is basically a definition of services that any concrete execution layer should offer. The second class in this package is *ExecuterViaXgird*, which is a concrete subclass of the abstract superclass *Executer*. It is implemented using the Mac OS X dependent Xgrid technology. Some example services that a concrete subclass of *Executer* should implement are: *submitLoopJob(), getDateSubmitted(), getDateStarted()*. For instance, the *submitLoopJob()* method is implemented in *ExecuterViaXgird* by sending a specific process using the xgrid command:

xgrid -h hostname -p password -job submit loopProcess arg1 arg2

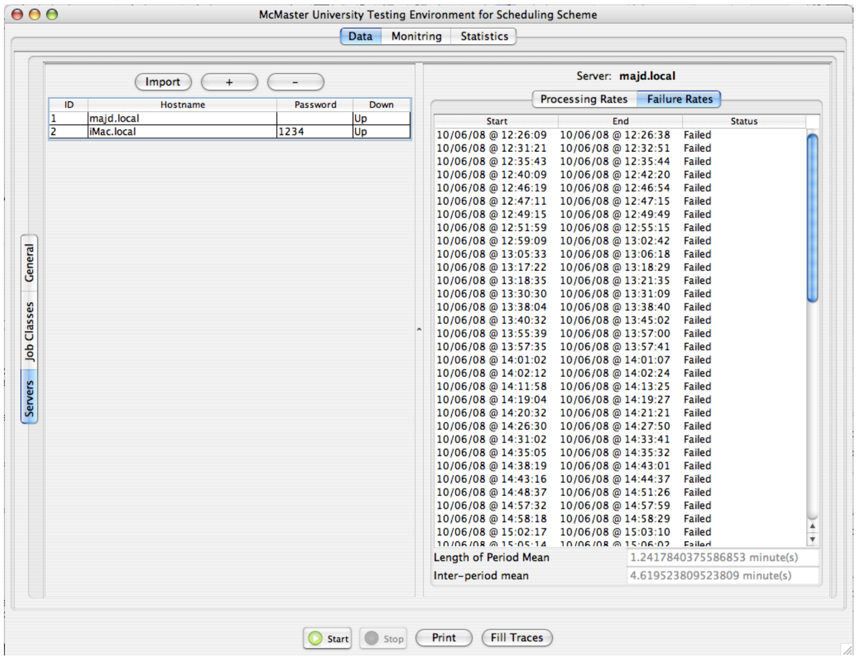
where *hostname* is the target machine address, *password* is the Xgrid password for that machine, *loopProcess* is an executable process and *arg1* and *arg2* are arguments for that process. The current implementation uses Xgrid technology for the jobs submission and execution management. However, the Xgrid technology can be substituted with another software layer. This can be done by implementing a new concrete class of the abstract class *Executer*.

6.3.3 generating

This package contains two classes. The first one is *JobsGenerator*. Each instance is responsible for the generation of jobs of one class according to some probability distribution. One instance of this class is associated with two instances of probability distribution classes (e.g. the exponential and uniform distributions). It uses one of them to calculate the periods between generation events to maintain an arrival rate under a specific distribution, and uses the other object to calculate the length of the jobs to create a variation in the length of jobs from the same class. After creating the jobs, they are sent to the active mapper instance. The second class in this package is *GeneratorsController*. The generators controller is responsible for controlling all of the instances of *JobsGenerators*. It initializes, starts and stops them.

6.3.4 interfacing

All the graphical user interface classes are contained in this package. The Java Swing library is used in this implementation. All other packages are completely independent



ID	Hostname	Password	Down
1	majd.local		Up
2	iMac.local	1234	Up

Server: **majd.local**

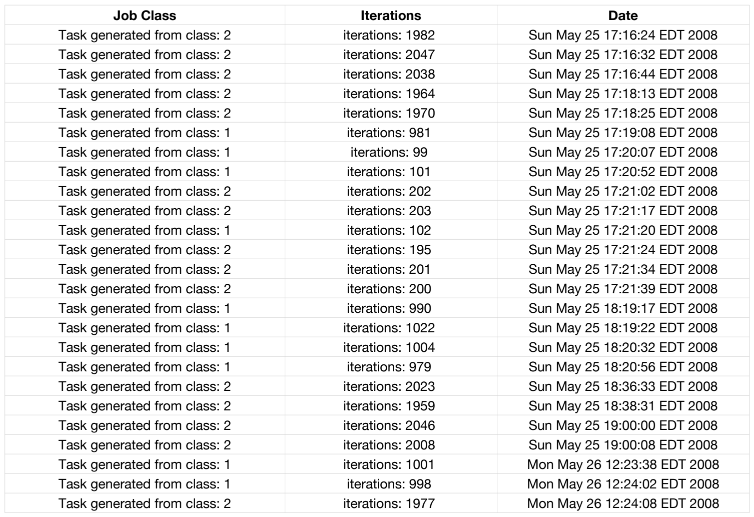
Start	End	Status
10/06/08 @ 12:26:09	10/06/08 @ 12:26:38	Failed
10/06/08 @ 12:31:21	10/06/08 @ 12:32:51	Failed
10/06/08 @ 12:35:43	10/06/08 @ 12:35:44	Failed
10/06/08 @ 12:40:09	10/06/08 @ 12:42:20	Failed
10/06/08 @ 12:46:19	10/06/08 @ 12:46:54	Failed
10/06/08 @ 12:47:11	10/06/08 @ 12:47:15	Failed
10/06/08 @ 12:49:15	10/06/08 @ 12:49:49	Failed
10/06/08 @ 12:51:59	10/06/08 @ 12:55:15	Failed
10/06/08 @ 12:59:09	10/06/08 @ 13:02:42	Failed
10/06/08 @ 13:05:33	10/06/08 @ 13:06:18	Failed
10/06/08 @ 13:17:22	10/06/08 @ 13:18:29	Failed
10/06/08 @ 13:18:35	10/06/08 @ 13:21:35	Failed
10/06/08 @ 13:30:30	10/06/08 @ 13:31:09	Failed
10/06/08 @ 13:38:04	10/06/08 @ 13:38:40	Failed
10/06/08 @ 13:40:32	10/06/08 @ 13:45:02	Failed
10/06/08 @ 13:55:39	10/06/08 @ 13:57:00	Failed
10/06/08 @ 13:57:35	10/06/08 @ 13:57:41	Failed
10/06/08 @ 14:01:02	10/06/08 @ 14:01:07	Failed
10/06/08 @ 14:02:12	10/06/08 @ 14:02:24	Failed
10/06/08 @ 14:11:58	10/06/08 @ 14:13:25	Failed
10/06/08 @ 14:19:04	10/06/08 @ 14:19:27	Failed
10/06/08 @ 14:20:32	10/06/08 @ 14:21:21	Failed
10/06/08 @ 14:26:30	10/06/08 @ 14:27:50	Failed
10/06/08 @ 14:31:02	10/06/08 @ 14:33:41	Failed
10/06/08 @ 14:35:05	10/06/08 @ 14:35:32	Failed
10/06/08 @ 14:38:19	10/06/08 @ 14:43:01	Failed
10/06/08 @ 14:43:16	10/06/08 @ 14:44:37	Failed
10/06/08 @ 14:48:37	10/06/08 @ 14:51:26	Failed
10/06/08 @ 14:57:32	10/06/08 @ 14:57:59	Failed
10/06/08 @ 14:58:18	10/06/08 @ 14:58:29	Failed
10/06/08 @ 15:02:17	10/06/08 @ 15:03:10	Failed
10/06/08 @ 15:05:14	10/06/08 @ 15:06:02	Failed

Length of Period Mean	1.2417840375586853 minute(s)
Inter-period mean	4.619523809523809 minute(s)

Figure 6.1: A screen shot showing a server table (left) and a failure trace view (right).

of this package. A Model View Controller (MVC) design pattern is used. MVC is a software design pattern used when designing user-interface based software. In an MVC panel, data classes (or models) are graphically represented by graphical classes (or views) and the data classes are manipulated and controlled by classes of a third type (controllers). Sometimes the controller and the view are combined in one class which is able to view and control the data model. Classes in this package represent the view and controller of data classes from other packages. In addition, the Java Event Delegation mechanism described earlier is used in this package.

This package contains more than 15 classes. One such class is named *JobClassesTableJPanel*. This class contains a table that provides a view of the job classes in the system. Through this view the user can change properties of any job class in the table. Other similar classes are *ServersTableJPanel* and *FailureTraceTable*, which in turn provide views of and control the servers' table and failure traces respectively (see Figure 6.1).



Job Class	Iterations	Date
Task generated from class: 2	iterations: 1982	Sun May 25 17:16:24 EDT 2008
Task generated from class: 2	iterations: 2047	Sun May 25 17:16:32 EDT 2008
Task generated from class: 2	iterations: 2038	Sun May 25 17:16:44 EDT 2008
Task generated from class: 2	iterations: 1964	Sun May 25 17:18:13 EDT 2008
Task generated from class: 2	iterations: 1970	Sun May 25 17:18:25 EDT 2008
Task generated from class: 1	iterations: 981	Sun May 25 17:19:08 EDT 2008
Task generated from class: 1	iterations: 99	Sun May 25 17:20:07 EDT 2008
Task generated from class: 1	iterations: 101	Sun May 25 17:20:52 EDT 2008
Task generated from class: 2	iterations: 202	Sun May 25 17:21:02 EDT 2008
Task generated from class: 2	iterations: 203	Sun May 25 17:21:17 EDT 2008
Task generated from class: 1	iterations: 102	Sun May 25 17:21:20 EDT 2008
Task generated from class: 2	iterations: 195	Sun May 25 17:21:24 EDT 2008
Task generated from class: 2	iterations: 201	Sun May 25 17:21:34 EDT 2008
Task generated from class: 2	iterations: 200	Sun May 25 17:21:39 EDT 2008
Task generated from class: 1	iterations: 990	Sun May 25 18:19:17 EDT 2008
Task generated from class: 1	iterations: 1022	Sun May 25 18:19:22 EDT 2008
Task generated from class: 1	iterations: 1004	Sun May 25 18:20:32 EDT 2008
Task generated from class: 1	iterations: 979	Sun May 25 18:20:56 EDT 2008
Task generated from class: 2	iterations: 2023	Sun May 25 18:36:33 EDT 2008
Task generated from class: 2	iterations: 1959	Sun May 25 18:38:31 EDT 2008
Task generated from class: 2	iterations: 2046	Sun May 25 19:00:00 EDT 2008
Task generated from class: 2	iterations: 2008	Sun May 25 19:00:08 EDT 2008
Task generated from class: 1	iterations: 1001	Mon May 26 12:23:38 EDT 2008
Task generated from class: 1	iterations: 998	Mon May 26 12:24:02 EDT 2008
Task generated from class: 2	iterations: 1977	Mon May 26 12:24:08 EDT 2008

Figure 6.2: A log file opened in Numbers software

6.3.5 logging

The classes of this package are responsible for the operation of logging system events. This package can be modified to use different schemes of storage. For simplicity and practicality the current implementation uses files. The text files produced can be opened and processed using spread sheet applications. This feature allows the tester to further study the results of their tests (Figure 6.2).

This package contains three classes. The class *Event* represents a generic event. The class *Logger* is responsible for communication with the storage layer (e.g file system or DBMS) and storing events with their time stamps. The third class is named *ServersReader* and it can restore information about *Servers* objects stored in a file. This allows testers to maintain a list of servers in a text file.

6.3.6 mapping

This package has several classes. The following is a partial list:

1. **AvailabilityServer** is a server that keeps listening (by default on port 37933) for availability updates sent by machines in the grid and consequently modifies the mapper information.

2. **CompletionServer** is a class that represents a server that keeps listening (by default on port 37931) for notifications of job completion and consequently modifies the mapper information.

3. **TimeoutThread**. A thread from this class is responsible for sending a notification to the mapper when a time-out occurs. It keeps track of all sent jobs. If a sent job was not completed by a certain time this thread will announce this job as Timed Out. This feature can be turned off.

4. **MappingScheme** is an abstract class. It contains some abstract methods that must be implemented by any concrete mapping scheme. More details on this class can be found in Section 6.4.

5. **Mapper** is a concrete class. One instance of this class co-operates with an instance of a concrete subclass of *MappingScheme* to perform the mapping operation. The *Mapper* class performs all of the general operations of mapping such as receiving a job and later sending it to a machine. The concrete *MappingScheme* (e.g. LPAS_DG) on the other hand, performs the mapping scheme-specific operations such as the actual scheduling and how time-outs are handled. The *Mapper* does not invoke different methods for each mapping scheme. It invokes the method defined in the abstract class *MappingScheme* and the the proper method is chosen using the polymorphism and dynamic binding features.

6. **LPAS_DG__MS** is a concrete class of *MappingScheme* that implements the LPAS_DG mapping scheme.

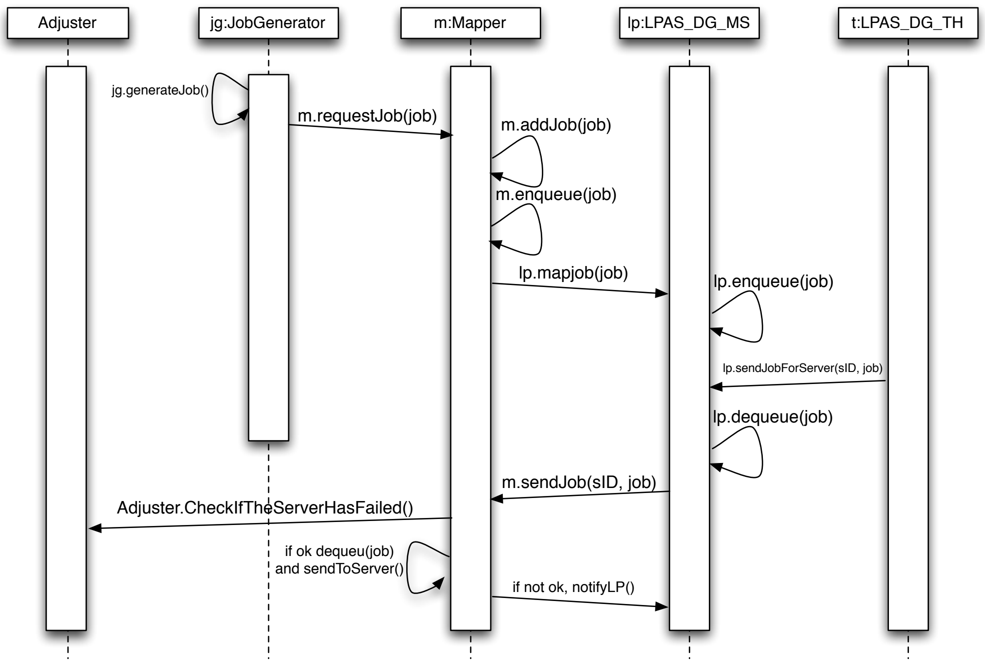


Figure 6.3: Sequence diagram showing communications when a job is sent and mapped.

7. **LPAS_DG_TH** is a thread that is part of the LPAS_DG implementation. This thread keeps checking the queues of jobs and sends jobs to an object of the *LPAS_DG_MS* class. The thread determines the order in which the available servers are used in the mapping process. In other words, this thread is responsible for choosing a server from the pool of available servers and notifies an object of class *LPAS_DG_MS* which in turn chooses a job to be sent to that server.

In Figure 6.3 the communications between objects of classes of this package during the process of mapping a job are shown.

6.3.7 pulling

The objects of this package's classes will run on the machines and not the mapper. This package is constructed from more than 12 classes. The following is a list of the important classes in this package:

1. **CompletionAnnouncer** is a class from which every machine has one object. That object is responsible for notifying the mapper when the execution of a job is completed and hence it needs to know the address of the mapper. The mapper in turn notifies this object when a job is sent to its machine.

2. **CPU_Eater** is used to make machines busy. In some tests the tester wishes to simulate some availability conditions. This thread is able to make one core in a CPU busy to a certain percentage determined by the user.

3. **AvailabilityManager** is a thread responsible for measuring CPU usage or CPU availability on machines and then logging and sending the results to the mapper. It is part of an availability prediction module developed as part of the environment.

4. **AvailabilityLogger** is used by the *AvailabilityThread* class to log the availability readings.

6.3.8 probability_distribution

This package has the classes related to probability distributions. It contains the following classes:

1. **ProbabilityDist** is an abstract class with one abstract method *getNextValue()*. This method should be defined in a manner such that each invocation produces a sample from a certain probability distribution.

2. **ExponentialDist** is a concrete subclass of *ProbabilityDist* that implements an exponential distribution.

3. **UniformDist** is a concrete subclass of *ProbabilityDist* that implements a uniform distribution.

6.4 Adding New Scheduling Policies

To add a mapping scheme, a new class must be added to the package *mapping*. This class must be concrete and must extend the abstract class *MappingScheme*, thus it must implement the methods of the abstract class *MappingScheme*. The methods are:

1. **public abstract void startMappingScheme()**. In this method, initialization operations are defined. For instance, if the mapping scheme depends on a thread, the thread is initialized and started.

2. **public abstract void stopMappingScheme()**. In this method, the programmer should define operations that stop the execution of the system processes (e.g mapping and generation). This can be useful if the system execution is wished to be restarted after stopping it.

3. **protected abstract void mapJob(Job job)**. The implementation of this method determines the mapping scheme. It is invoked by the mapper object. The job is sent to the active *MappingScheme* concrete object (e.g. the LPAS_DG object or the MET object). This object then makes the mapping decision according to the appropriate policy.

4. **public abstract void handleJobTimeOut(long jobID).** The implementation of this method determines what should happen when a job times out. One way of handling a time-out for instance, is to resubmit the timed-out job. This way was chosen for the implemented schemes.

5. **public abstract void serverIsDown(int serverID)** The implementation of this method determines what should happen when a server goes down. For instance, the LPAS_DG policy re-solves the LP allocation.

6. **public abstract void serverIsUp(int serverID)** The implementation of this method determines what should happen when a server becomes up after being down. For instance, the LPAS_DG policy re-solves the LP allocation.

Three mapping schemes are implemented. These are LPAS_DG, Gcμ and FCFS. Classes are named *LPAS_DG_MS*, *Gcu_MS* and *FCFS_MS*. The convention is to append the name of the mapping scheme with "_MS". These implementations can be found in the source code on the attached Compact Disc (Appendix A).

6.5 Adding New Probability Distributions

Adding a new probability distribution requires adding a new concrete class to the *probabilityDist* package that extends the abstract class *ProbabilityDist*. The single method that has to be implemented is the method with the signature *public abstract double getNextValue()*. For instance to implement the exponential distribution the method is implemented as follows:

*return (-1*Math.log(Math.random()))/this.getLambda();*
or
$-log(r)/\lambda$ *(where r is a random number in* $[0, 1)$ *and* $1/\lambda$ *is the mean)*

Invoking the method above will produce a sample from an exponential distribution with mean $1/\lambda$. The exponential and uniform distributions were implemented in the *ExponentialDist* and *UniformDist* classes respectively. These implementations can be found in the source code on the attached Compact Disc (Appendix A).

6.6 Considerations in Design and Development

During the first phase of development the problem domain was analysed and the workload model of the theoretical scheme, which the system is supposed to test, was studied. Upon understanding the problem domain, design requirements were determined. Different aspects were considered, each of which is discussed in detail in the reminder of this section.

Maintainability - A software system is said to be maintainable if it can be modified to adopt new requirements or to fix errors in a fluent manner. The maintainability aspect was a prime consideration. The main reason for this is that the requirements of our system are dynamic. The workload and the problem domain are well defined for the current research, however, the model may change in the future. Therefore, classes were designed with future additions in mind. The whole architecture of the system is suitable for maintenance and adding new features. In addition, the code was well documented using Javadoc [29]. (Javadoc is a tool for generating API documentation in HTML format from comments in source code.) Moreover, the source code was extensively commented.

Modularity - Every software system has several functionalities. Building a software system involves designing the internal modules of the system and defining how these modules interact. Generally speaking, each component should be independent of the other components to the largest extent possible. However, these components have to know how to communicate with each other and therefore they are not fully independent. The less dependent the modules are, the easier the system is to maintain. In order to achieve high modularity, each component was designed to have a specific and defined functionality. Each system component is implemented as a Java package and each package can access specific services from other packages. As an example all the mapping functionality is encapsulated in the *Mapper* module, whereas the logging functionality is encapsulated in the *Logger* module.

Extensibility - Extensibility is the ability to extend the system's features or functionality. The system was designed with extensibility at the top of the design requirements list. As previously mentioned, the system's main purpose is testing scheduling schemes. To do this, the system will support some scheduling schemes such as LPAS_DG. Adding additional scheduling schemes requires only inheriting an abstract class and implementing a set of methods. More details were given in Section 6.4. For example, the policies LPAS_DG, G$c\mu$ and FCFS were added using the method described in Section 6.4.

Security - The security aspect of the system was not a big concern in the design process, since the system will run in an academic environment. However, that does not mean that the system is not secure. No one can request jobs from the system without a password set in the configuration phase. For the purpose of testing, this is what really matters, that our Xgrid agents (servers) will not be receiving jobs to work on from outside entities.

Compatibility - The testing environment is to be installed on departmental machines. The system is constructed from two main software components. The main one is written in Java thus guaranteeing compatibility on different platforms. The other component (Xgrid technology) is Mac OS dependent. We decided that the system should work on both Mac OS X 10.4 (Tiger OS) and Mac OS X 10.5 (Leopard OS) and the system was tested on both operating systems. We expect that the testing environment will work well on future Apple platforms, but we cannot guarantee it. However, if major modifications were introduced to the Xgrid system, only one layer of our system would need to be modified, the Xgrid dependent layer.

Robustness - A software application is robust if it is able to tolerate unpredictable or invalid inputs or conditions. To achieve robustness, at the design level we tried to eliminate unpredictable conditions by simplifying messages between different modules. At the implementation level we always tried to check for different boundary conditions. In addition, the Java error handling model was extensively employed to catch exceptions. For example, all communications in the system were subject to time-out exceptions and the Xgrid system messages were always verified and checked, with an exception raised in case of a problem. Moreover, most of the system components are multi-thread based. As a result, every data structure was chosen from the java.util package to be thread safe, meaning that synchronization methods are used to guarantee the integrity of the thread-accessed data structures, thus protecting the integrity of the system state. In addition, no deprecated unsafe threads methods were used. Finally, it is worth mentioning that the Java programming language is robust. As a matter of fact, robustness of software created by Java is a main concern for Java designers [25]. The Xgrid technology is also robust and it is being used in large scale projects such as the Xgrid@Stanford project [49].

Chapter 7

Analysis

7.1 Introduction

In this chapter we will discuss some experiments conducted on our testing environment and will compare them to results obtained from the simulation tool used in [3]. Each of the following sections discusses one setting of servers and job classes and the experiments conducted using this setting. Although the description in each section may seem a bit repetitive, we have described each experiment in detail so that the reader can read the results of one experiment independently of the remainder of the chapter.

As in Section 3.1, α is the arrival rate vector of job classes, where the i^{th} element α_i, is the arrival rate of job class i. Moreover, the execution rate that a machine j can execute a job from class i is denoted by $\mu_{i,j}$. The availability of machine j is donated by a_j and the actual execution rate is given by $\mu'_{i,j} = \mu_{i,j} a_j$. In addition, μ_i is a vector that represents the execution rates for a particular job class, with the j^{th} element in this vector being $\mu_{i,j}$. Finally, μ is the matrix constructed by all execution rate vectors, where entry (i, j) is $\mu_{i,j}$.

The metrics used in the simulations and experiments are the average waiting time and the average completion time (response time). For the experimental part we also used the average communication delay.

As a reminder, the waiting time is the difference between the time that a job is submitted and is sent to a server. The response time is the difference between the time a job is submitted and the time it completes execution. The communication delay is the difference between the time a job is sent to be executed and the time it begins execution. This delay occurs mainly due to communication delays, but it could also be caused by the software layer responsible for the distribution and execution of the tasks.

Machine heterogeneity refers to the average variation in the rows of the execution matrix μ. Similarly, job heterogeneity refers to the average variation of the columns. Based on this and following [10], we define the following categories for heterogeneity:

- High job heterogeneity and high machine heterogeneity (*HiHi*)
- High job heterogeneity and low machine heterogeneity (*HiLo*)
- Low job heterogeneity and high machine heterogeneity (*LoHi*)
- Low job heterogeneity and low machine heterogeneity (*LoLo*)

Every setting from the following belongs to one of the above categories.

7.2 Setting ***HiHi***

This setting was constructed from 6 machines and 4 job classes. The LPAS_DG, Gcμ and FCFS policies were simulated and tested on this setting.

- Execution rates are shown in Table 7.1. M1 to M6 are machine 1 to machine 6.

Class	M1	M2	M3	M4	M5	M6
1	2.0	2.0	2.0	2.0	2.0	2.0
2	1.0	20.0	3.7	7.1	2.4	8.7
3	1.0	20.0	9.4	3.7	7.2	2.7
4	1.0	20.0	2.8	5.9	4.4	6.3

Table 7.1: Execution Rates of Setting A

- The arrival rates of the job classes were

$$\alpha = \left[\begin{array}{cccc} 2.25 & 4.50 & 7.20 & 12.60 \end{array} \right].$$

7.2.1 Experiment 1

This experiment was conducted on Setting *HiHi* with the following parameters:

- All machines were dedicated ($a_j = 1.0, \forall j$).
- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.2. The metric used in the table is the mean response time. The confidence level was 95%. The confidence intervals are shown between brackets in the table.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.56, 0.57)	(0.66, 0.67)	(1.30, 1.33)
2	(0.33, 0.34)	(0.26, 0.26)	(0.99, 1.02)
3	(0.18, 0.18)	(0.25, 0.25)	(0.99, 1.02)
4	(0.11, 0.11)	(0.27, 0.27)	(0.99, 1.02)
Overall	(0.20, 0.21)	(0.30, 0.30)	(1.01, 1.05)

Table 7.2: Results of simulation 1

Our testing environment was used to conduct the experiment, and the results are shown in Table 7.3. At the time when the experiment was stopped the actual arrival rates of job classes were within 5% of the assumed arrival rates α. The LPAS_DG test took 90 minutes (45 time units), while the $Gc\mu$ and FCFS tests took 10.5 hours (315 time units) and 21 hours (253 time units), respectively.

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.58	0.78	1.27
2	0.34	0.29	1.01
3	0.22	0.21	0.98
4	0.17	0.31	0.97
Overall	0.24	0.34	1.00

Table 7.3: Results of experiment 1

Discussion

In Figure 7.1 a comparison between the results of the simulation and the results of the experiment is shown. The left side of the figure shows the simulation results, whereas the right side shows the experimental results. The results were similar. All of the average response times obtained by the experiment were within 0.06 time units of the corresponding entries in the simulation table.

The results of this experiment verified the results of [3], that is the superiority in the performance of the LPAS_DG policy over the Gcμ and FCFS policies. Both the simulation and the experimental results show an increase of performance of 4 - 5 times when using the LPAS_DG policy instead of the FCFS policy in heterogeneous environments. These results also show that the abstract model assumed by [3] is reasonable and that the unmodeled overhead of LPAS_DG in this case had minimal impact, since the LP allocation problem was solved only once and the communication delay was not large (since the experiment was conducted in the same Local Area Network (LAN)).

7.2.2 Experiment 2

This experiment was conducted on Setting *HiHi* with the following parameters:

- $$a_j = \begin{cases} 1.0 & \text{if } 1 \leq j \leq 3 \\ 0.5 & \text{if } 4 \leq j \leq 6 \end{cases}$$
- This setting included no machine failures.

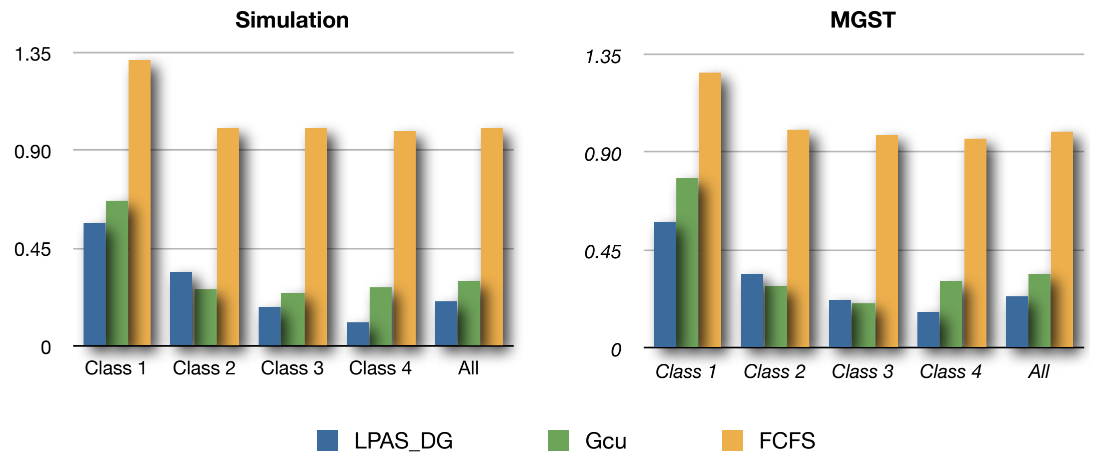


Figure 7.1: Experiment 1 results.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.4. The metric used here is the mean response time. The confidence level was 95%. FCFS is not stable for this setting and parameters.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.97, 0.98)	(1.10, 1.11)	N/A
2	(0.22, 0.22)	(0.42, 0.42)	N/A
3	(0.27, 0.27)	(0.36, 0.36)	N/A
4	(0.16, 0.16)	(0.44, 0.44)	N/A
Overall	(0.27, 0.27)	(0.47, 0.47)	N/A

Table 7.4: Results of simulation 2

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.5.

LPAS_DG Test. This test took 6.5 hours (130.7 time units). The actual arrival rate was

$$\alpha' = \begin{bmatrix} 2.17 & 4.60 & 7.19 & 11.95 \end{bmatrix},$$

which is within 5% of the desired arrival rate. The actual execution rates are given

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.99	1.51	N/A
2	0.37	0.59	N/A
3	0.29	0.50	N/A
4	0.43	0.59	N/A
Overall	0.42	0.64	N/A

Table 7.5: Overall results of Experiment 2

in the following table:

Class	M1	M2	M3	M4	M5	M6
1	1.96	N/A	N/A	0.96	0.94	N/A
2	N/A	18.04	N/A	3.27	N/A	3.90
3	N/A	N/A	8.70	N/A	3.30	N/A
4	N/A	18.09	N/A	N/A	N/A	N/A

Table 7.6: Execution Rates in LPAS_DG test in Experiment 2

Gcμ Test. This test took 3.9 hours (81.7 time units). The actual arrival rate

$$\alpha' = \begin{bmatrix} 2.22 & 4.90 & 7.12 & 12.57 \end{bmatrix},$$

which is within 10% of the actual desired rate. The actual execution rates are given in the following table:

Class	M1	M2	M3	M4	M5	M6
1	1.92	2.05	2.03	0.90	0.94	0.92
2	1.01	18.04	3.66	3.23	1.10	3.93
3	0.98	17.92	8.70	1.68	3.30	1.27
4	1.01	18.09	2.80	2.72	2.00	2.84

Table 7.7: Execution Rates in Gcμ test in Experiment 2

FCFS Test

This test took 17.7 hours (500 time units). The actual arrival rate was

$$\alpha' = \begin{bmatrix} 2.24 & 4.13 & 6.84 & 12.41 \end{bmatrix},$$

which is within 10% of the desired arrival rate.

The experiment showed that this policy is unstable. Both the queue of waiting jobs and the response time were growing with time. Figure 7.2 shows the relation between time (in time units) and response time (in time units).

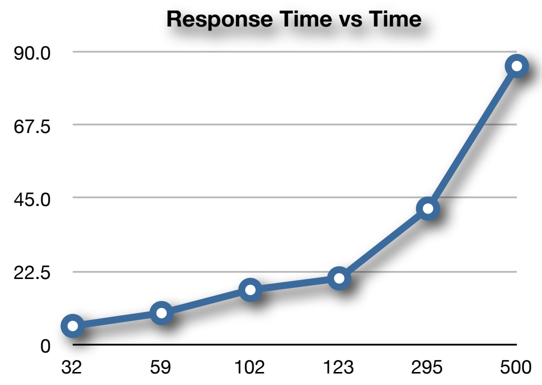


Figure 7.2: Experiment 2, FCFS test results.

Discussion

The FCFS policy test on both the simulation tool and the testing environment showed that the FCFS policy is unstable. As such, the LPAS_DG policy proves to be superior. It is worth noting though that since the actual processing rates (Table 7.6 and Table 7.7) are slower than the assumed processing rates (Table 7.1), the response times of the experiment were larger than those of the simulation.

Additionally, the performance of LPAS_DG in processing class 4 was much slower in the test than in the simulation. We believe that the reason behind this is that according to the δ^* matrix (Section 4.4.6), which results from solving the LP, class 4 jobs can only be processed by machine 4. The actual rate $\bar{\mu}_{4,4}$ was 18.09, whereas the desired $\mu_{4,4}$ was 20. The fact that only one machine can execute jobs from this class makes the performance of the policy highly dependent on that machine. In this experiment the machine could not reach the ideal execution rate, which resulted in poor performance for that class. The fact that only one or few machines can execute particular job classes makes the performance of the policy very sensitive to the performance of these machines. If these machines under-performed (due to over estimation of execution rates or machine failures), the performance of the policy will deteriorate. In large grids however, these groups are constructed from a large number of machines which should attenuate this effect. Nevertheless, we think that this downside of the LPAS_DG policy warrants further examination.

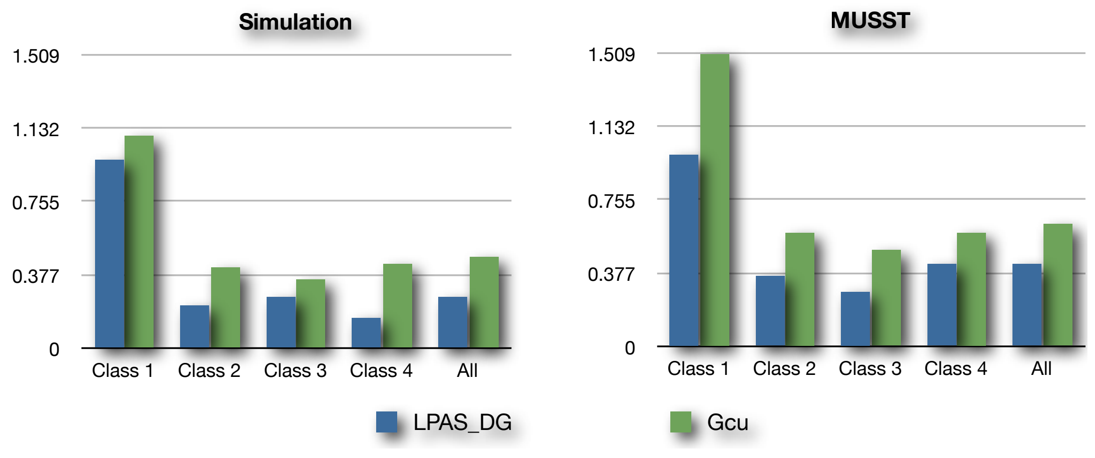


Figure 7.3: Experiment 2 results.

7.2.3 Experiment 3

This experiment was conducted on Setting *HiHi* with the following parameters:

- a_j =1.0 (for all j)
- Each machine fails at the rate 0.05 per time-unit and the mean fault time is 2 time-units. The periods were exponentially distributed.

Results

The simulations were conducted using the simulation software used in [3] and the results obtained are shown in Table 7.8. The metric used here is the mean response time. The confidence level was 95%. FCFS is not stable for this setting and parameters.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.61, 0.61)	(0.73, 0.73)	N/A
2	(0.35, 0.35)	(0.28, 0.28)	N/A
3	(0.19, 0.20)	(0.27, 0.27)	N/A
4	(0.13, 0.13)	(0.30, 0.30)	N/A
Overall	(0.23, 0.23)	(0.32, 0.33)	N/A

Table 7.8: Results of simulation 3

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.9. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 2%, 8% and 5% respectively of the assumed arrival rates. The LPAS_DG test took 12.9 hours (258.7 time units), while Gcμ took 3.8 hours (56.4 time units).

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.61	0.77	N/A
2	0.45	0.30	N/A
3	0.20	0.26	N/A
4	0.15	0.30	N/A
Overall	0.25	0.33	N/A

Table 7.9: Overall results of experiment 3

The experiment showed that FCFS is unstable. Both the queue of waiting jobs and the response time were growing with time. Figure 7.2 shows the relation between time (in time units) and response time (in time units).

Discussion

In Figure 7.5 a comparison between the results of the simulation and the results of the experiment is shown. The left side of the figure shows the simulation results, whereas the right side shows the experimental results. The results were similar. All

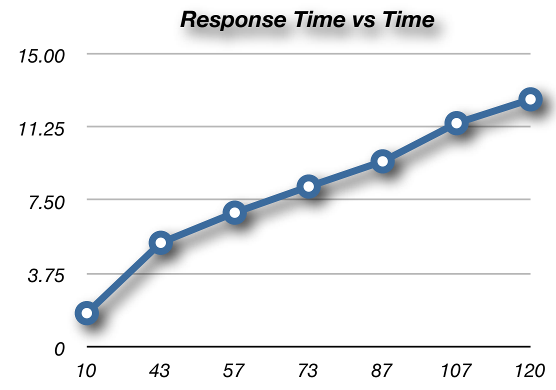


Figure 7.4: Experiment 3, FCFS test results.

the response times obtained by the experiment were within 0.1 time units of the corresponding entries in the simulation table.

The results of this experiment verified the results of [3], that is the superiority in the performance of the LPAS_DG policy over the Gcμ and FCFS policies.

These results also show that the abstract model assumed by [3] is reasonable and that the impact of the overhead of LPAS_DG in this case was minimal since the LP allocation problem was solved only once. In addition, the communication delay was minimal since the communications between the machines happened in the same LAN.

7.2.4 Experiment 4

This experiment was conducted on Setting *HiHi* with the following parameters:

- $$a_j = \begin{cases} 1.0 & \text{if } 1 \leq j \leq 3 \\ 0.5 & \text{if } 4 \leq j \leq 6 \end{cases}$$

- Each machine fails at the rate 0.05 per time-unit and the mean fault time is 2 time-units. The periods were exponentially distributed.

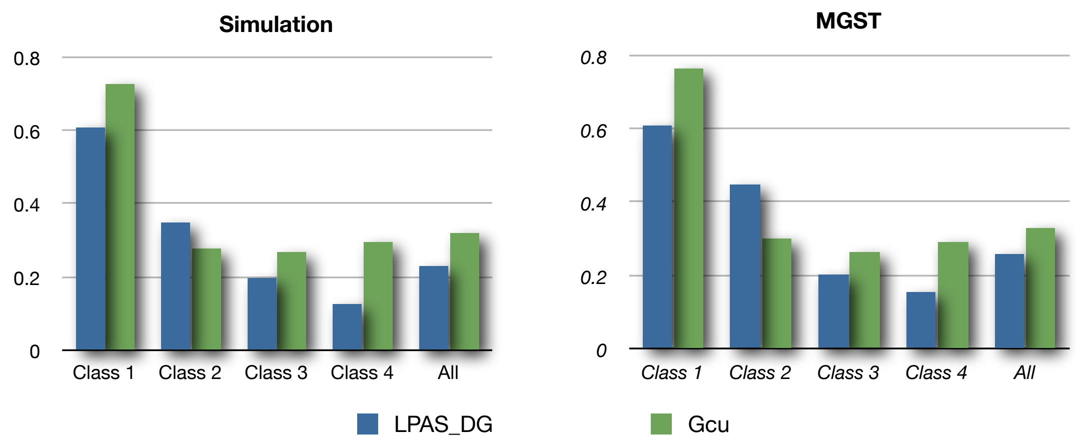


Figure 7.5: Experiment 3 results.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.10. The metric used here is the mean response time. The confidence level was 95%. FCFS is not stable for this setting and parameters.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(1.10, 1.11)	(1.26, 1.27)	N/A
2	(0.30, 0.31)	(0.50, 0.51)	N/A
3	(0.32, 0.32)	(0.42, 0.42)	N/A
4	(0.26, 0.27)	(0.53, 0.53)	N/A
Overall	(0.35, 0.36)	(0.56, 0.56)	N/A

Table 7.10: Results of simulation 4

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.11. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 7%, 5% and 5% respectively of the desired arrival rates. The LPAS_DG test took 14.4 hours (309 time units), while the Gcμ took 2 hours (40 time units). The experiment showed that FCFS is unstable. Both the queue of waiting jobs and the response time were growing with time. Figure 7.6 shows the relation between time (in time units) and response time (in time units).

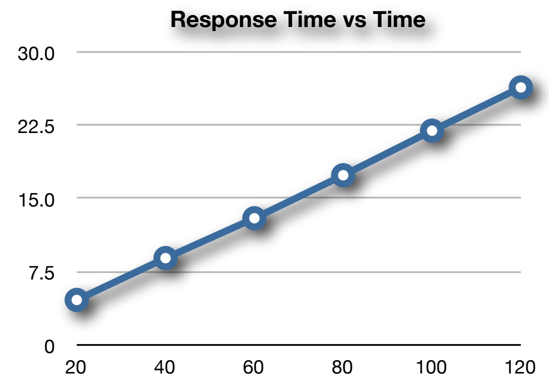


Figure 7.6: Experiment 4, FCFS test results.

Class	LPAS_DG	$Gc\mu$	FCFS
1	1.03	1.30	N/A
2	0.50	0.55	N/A
3	0.46	0.41	N/A
4	1.36	0.51	N/A
Overall	0.94	0.56	N/A

Table 7.11: Overall results of experiment 4

Discussion

The difference between the response time of class 4 in the simulation and the experiment is due to two factors.

The first one is the effective rate in the experiment is less than that assumed by LPAS_DG. The second and more important factor is that the delay between the completion time and notification time of completion is large compared to the execution time. This delay is usually negligible, but in this case it is large compared to the execution time. Jobs from class 4 are exclusive to machine 4 and the processing rate of machine 4 is large (20 jobs per time unit or an execution time of 0.05 time units). In the test, the time unit was 3 minutes, thus the processing time of this machine for class 4 was 9 seconds (1 / $\mu_{4,4}$ = 9 seconds). The actual processing rate was 10-11 seconds, add to that the delay time between the completion time and the notification time which was approximately 2 seconds (more than 10% of the desired processing time). This results in enlarging the actual effective execution rate by approximately 44%.

The entry (4,4) in the δ^* matrix was .936 and the ρ^* value was .673 which mean that this machine should be busy approximately 67% of time when it is up (without failures) in ideal conditions. Its busy time is divided between class 2 and class 4 jobs in a ratio of 6:94. Taking the effective execution rate into account the load on that machine rises to approximately 97% without failures. If the failure rate was taken into consideration the load would exceed 100% or equivalently the effective processing rate of the machine becomes slower than the arrival rate of the class 4, which results in class 4 becoming unstable. We believe that one class being exclusive to one job class might give higher than desired sensitivity to system parameters.

In addition, the response times of class 2 and class 3 are higher than the simulation, as the unavailability of machine 4 meant the other machines became more highly loaded.

The G$c\mu$ performance in this case was better than the LPAS_DG policy, indicating that it suffers less from the sensitivity problem.

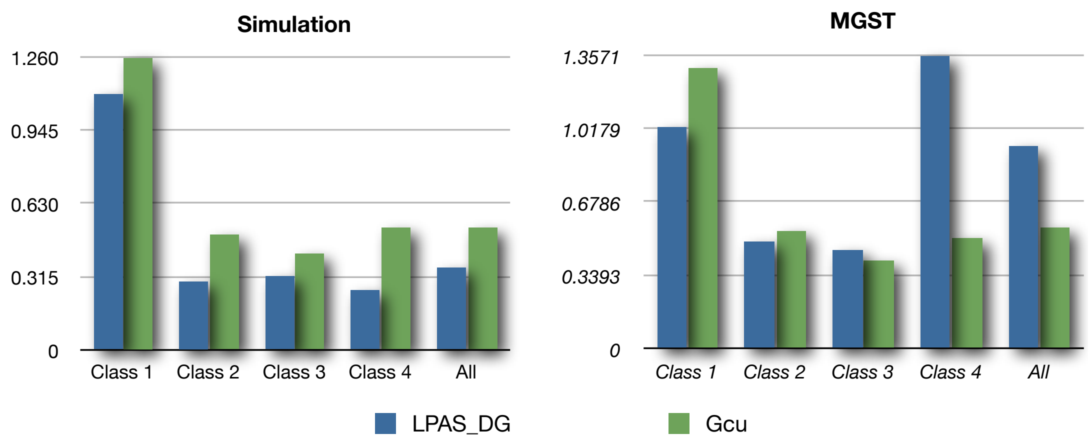


Figure 7.7: Experiment 4 results.

7.3 Setting *LoHi*

This setting was constructed from the category *LoHi*, and had 21 machines and 4 job classes. There were seven groups of machines. Members of the same group have the same execution rates. Machines in group 1 are machines 1, 8 and 15, machines in group 2 are machines 2, 9 and 16 etc. Formally, machine j belongs to group i if and only if j mod $7 = i$.

- Execution rates are shown in Table 7.12. G1 to G7 are group 1 to group 7.

Class	G1	G2	G3	G4	G5	G6	G7
1	2.20	7.00	10.25	1.00	5.70	0.50	12.00
2	1.95	7.05	9.78	0.95	5.65	0.56	11.85
3	2.00	7.25	10.02	0.98	5.75	0.67	11.80
4	2.05	6.75	9.99	1.02	5.82	0.49	12.05

Table 7.12: Execution Rates of Setting LoHi

- The arrival rates of the job classes were

$$\alpha = \begin{bmatrix} 22.5 & 22.5 & 18.0 & 18.0 \end{bmatrix}.$$

7.3.1 Experiment 5

This experiment was conducted on Setting *LoHi* with the following parameters:

- All machines were dedicated in this experiment ($a_j = 1, \forall j$).
- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.13. The metric used here is the mean response time. The confidence level was 95%.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.22, 0.22)	(0.21, 0.21)	(0.21, 0.21)
2	(0.12, 0.12)	(0.21, 0.22)	(0.21, 0.22)
3	(0.30, 0.30)	(0.21, 0.21)	(0.21, 0.21)
4	(0.29, 0.29)	(0.21, 0.21)	(0.22, 0.22)
Overall	(0.22, 0.22)	(0.21, 0.21)	(0.21, 0.21)

Table 7.13: Results of simulation 5

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.14. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 4%, 3% and 4% respectively of the assumed arrival rates. The LPAS_DG test took 100 minutes (50 time units), while the Gcμ and FCFS tests took 100 minutes (50 time units) and 320 minutes (160 time units) respectively.

Discussion

The results of the simulation and our testing environment were similar.

7.3.2 Experiment 6

This experiment was conducted on Setting *LoHi* with the following parameters:

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.31	0.26	0.26
2	0.22	0.25	0.26
3	0.37	0.24	0.25
4	0.35	0.25	0.26
Overall	0.31	0.25	0.26

Table 7.14: Overall results of Experiment 5

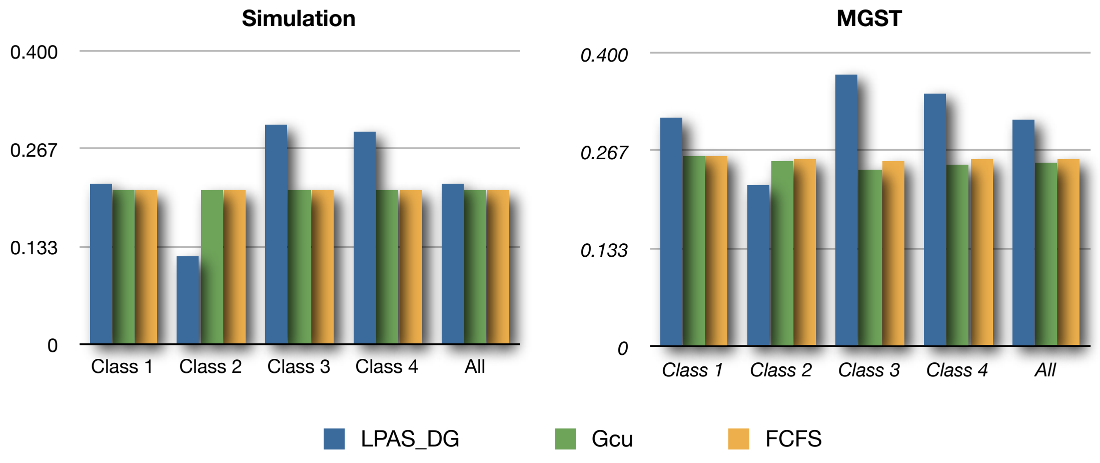


Figure 7.8: Experiment 5 results.

- The availabilities of machines were as follows:

$$a_j = \begin{cases} 0.50 & \text{if } j = 2,\ 11 \text{ or } 19 \\ 0.75 & \text{if } j = 3,\ 12 \text{ or } 20 \\ 1.00 & \text{otherwise} \end{cases}$$

- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.15. The metric used here is the mean response time. The confidence level was 95%.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.24, 0.24)	(0.23, 0.23)	(0.24, 0.24)
2	(0.13, 0.13)	(0.24, 0.24)	(0.24, 0.24)
3	(0.37, 0.37)	(0.23, 0.23)	(0.23, 0.23)
4	(0.35, 0.35)	(0.24, 0.24)	(0.24, 0.24)
Overall	(0.26 - 0.27)	(0.24, 0.24)	(0.24, 0.24)

Table 7.15: Results of simulation 6

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.16. At the time when the LPAS_DG, G$c\mu$ and the FCFS tests were stopped the actual arrival rates of job classes were within 6%, 8% and 5% respectively of the desired arrival rates. The LPAS_DG test took 110 minutes (55 time units), while both the G$c\mu$ and FCFS tests took 100 minutes (50 time units).

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.36	0.30	0.31
2	0.26	0.32	0.32
3	0.44	0.31	0.31
4	0.47	0.33	0.32
Overall	0.37	0.31	0.31

Table 7.16: Overall results of experiment 6

Discussion

The results of the simulation and our testing environment were similar.

7.4 Setting *HiLo*

This setting was constructed from 21 machines and 4 job classes. This setting was from category *HiLo*. There were seven groups of machines. Members of the same group have the same execution rates. Machines in group 1 are machines 1, 8 and 15, machines in group 2 are machines 2, 9 and 16, etc. Formally, machine j belongs to group i if and only if $j \bmod 7 = i$.

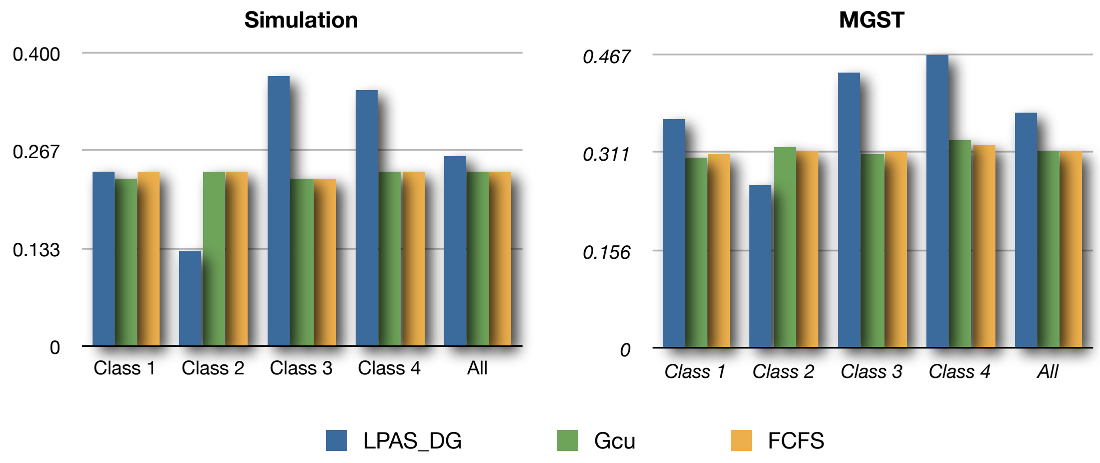


Figure 7.9: Experiment 6 results.

- Execution rates are shown in Table 7.17. G1 to G7 are group 1 to group 7.

Class	G1	G2	G3	G4	G5	G6	G7
1	2.00	2.50	2.25	2.00	2.20	1.75	2.25
2	4.50	4.0	4.20	4.00	3.80	3.90	3.95
3	6.00	6.20	6.25	6.00	5.75	5.90	6.05
4	10.00	10.25	10.50	9.50	10.25	10.25	10.00

Table 7.17: Execution Rates of Setting HiLo

- The arrival rates of the job classes were

$$\alpha = \begin{bmatrix} 10.50 & 21.00 & 26.25 & 26.25 \end{bmatrix}.$$

7.4.1 Experiment 7

This experiment was conducted on Setting *HiLo* with the following parameters:

- All machines were dedicated in this experiment ($a_j = 1, \forall j$).
- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.18. The metric used here is the mean response time. The confidence level was 95%.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.49, 0.49)	(0.50, 0.50)	(0.49, 0.49)
2	(0.28, 0.28)	(0.26, 0.26)	(0.27, 0.27)
3	(0.24, 0.24)	(0.18, 0.18)	(0.18, 0.18)
4	(0.14, 0.14)	(0.11, 0.11)	(0.12, 0.12)
Overall	(0.25, 0.25)	(0.22, 0.22)	(0.22, 0.22)

Table 7.18: Results of simulation 7

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.19. At the time when the LPAS_DG, G$c\mu$ and the FCFS tests were stopped the actual arrival rates of job classes were within 4%, 7% and 6%, respectively of the assumed arrival rates. All the tests took 100 minutes (50 time units).

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.50	0.53	0.51
2	0.31	0.30	0.31
3	0.32	0.21	0.23
4	0.35	0.14	0.17
Overall	0.35	0.25	0.26

Table 7.19: Overall results of experiment 7

Discussion

The results of the simulation and our testing environment were similar.

7.4.2 Experiment 8

This experiment was conducted on Setting *HiLo* with the following parameters:

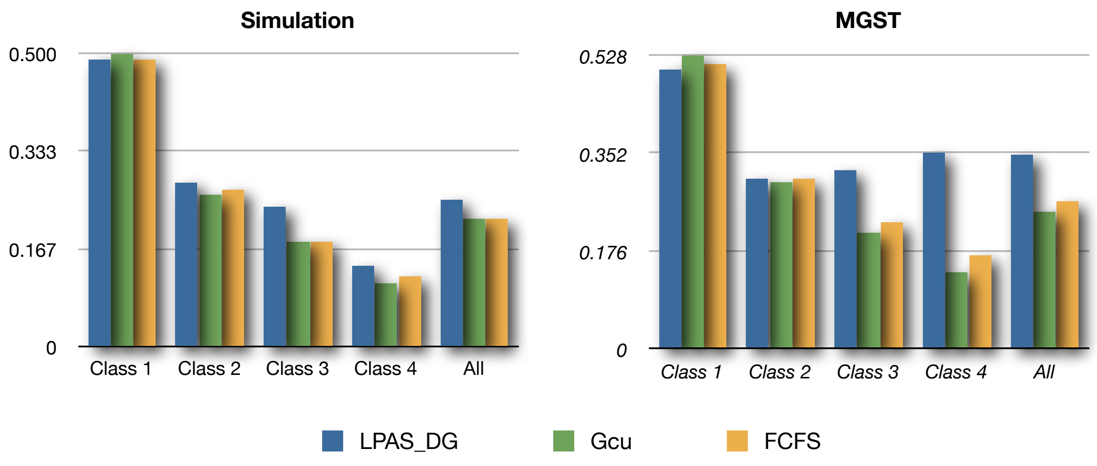


Figure 7.10: Experiment 7 results.

- The availabilities of machines were as follows:

$$a_j = \begin{cases} 0.50 & \text{if } j = 2, 11 \text{ or } 19 \\ 0.75 & \text{if } j = 3, 12 \text{ or } 20 \\ 1.00 & \text{otherwise} \end{cases}$$

- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.20. The metric used here is the mean response time. The confidence level was 95%.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.79, 0.80)	(0.64, 0.65)	(0.62, 0.62)
2	(0.42, 0.42)	(0.35, 0.35)	(0.37, 0.37)
3	(0.27, 0.27)	(0.24, 0.24)	(0.28, 0.28)
4	(0.19, 0.19)	(0.14, 0.15)	(0.20, 0.21)
Overall	(0.35, 0.35)	(0.29, 0.29)	(0.32, 0.32)

Table 7.20: Results of simulation 8

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.21. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 5%, 10% and 10% respectively of the assumed arrival rates. The LPAS_DG test took 124 minutes (62 time units), while the Gcμ and FCFS tests took 104 minutes (52 time units) and 100 minutes (50 time units) respectively.

Class	LPAS_DG	$Gc\mu$	FCFS
1	1.22	0.96	0.65
2	0.77	0.54	0.41
3	0.53	0.38	0.33
4	0.73	0.24	0.25
Overall	0.74	0.44	0.36

Table 7.21: Overall results of experiment 8

Discussion

Compared to the simulation, the LPAS_DG and GCμ policies performed poorly in the test. The reason is that the ideal overall load on the machines was fairly high (86.4%), but the different sources of errors and overhead caused the load to be close to 100%. The sources of errors are higher overall arrival rates, over estimation for processing rates and communication overhead coupled with the scheduling delay.

7.5 Setting *LoLo*

This setting was constructed from 21 machines and 4 job classes. This setting was from category *LoLo*. There were seven groups of machines. Members of the same group have the same execution rates. Machines in group 1 are machines 1, 8 and 15, machines in group 2 are machines 2, 9 and 16, etc. Formally, machine j belongs to group i if and only if j mod $7 = i$.

- Execution rates are shown in Table 7.22. G1 to G7 are group 1 to group 7.

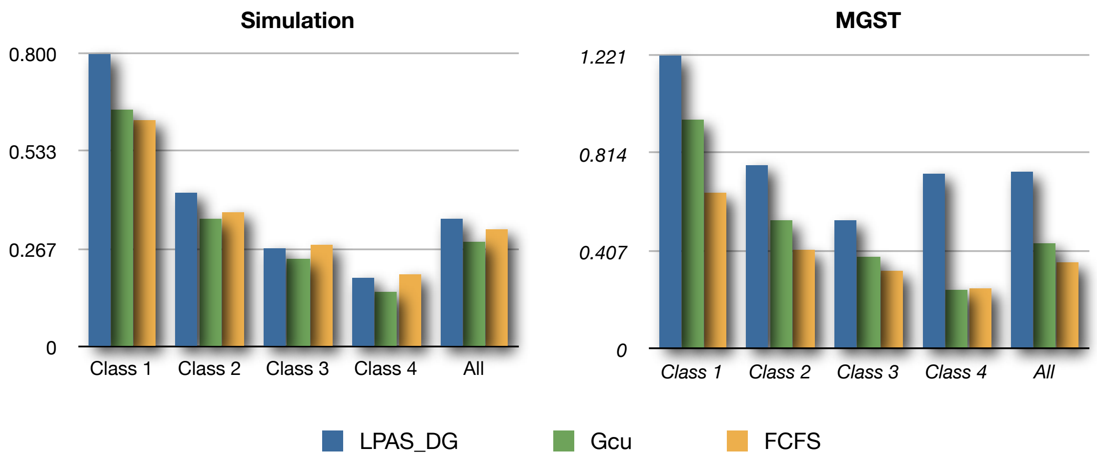


Figure 7.11: Experiment 8 results.

Class	G1	G2	G3	G4	G5	G6	G7
1	2.00	2.50	2.25	2	2.20	1.75	2.25
2	4.50	4.00	4.20	4	3.80	3.90	3.95
3	6.00	6.20	6.25	6	5.75	5.90	6.05
4	10.00	10.25	10.50	9.50	10.25	10.25	10.00

Table 7.22: Execution Rates of Setting LoLo

- The arrival rates of the job classes were

$$\alpha = \begin{bmatrix} 18.00 & 20.25 & 15.75 & 22.50 \end{bmatrix}.$$

7.5.1 Experiment 9

This experiment was conducted on Setting *LoLo* with the following parameters:

- All machines were dedicated in this experiment ($a_j = 1.0, \forall j$).
- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.23. The metric used here is the mean response time.

The confidence level was 95%.

Class	LPAS_DG	*Gcμ*	FCFS
1	(0.25, 0.25)	(0.20, 0.20)	(0.20, 0.20)
2	(0.23, 0.23)	(0.20, 0.20)	(0.20, 0.20)
3	(0.23, 0.23)	(0.21, 0.21)	(0.21, 0.21)
4	(0.21, 0.22)	(0.20, 0.20)	(0.20, 0.20)
Overall	(0.23, 0.23)	(0.21, 0.21)	(0.21, 0.21)

Table 7.23: Results of simulation 9

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.24. At the time when the tests were stopped the actual arrival rates of job classes were within 5% of the assumed arrival rates. The LPAS_DG test took 108 minutes (54 time units), while the Gcμ and FCFS tests took 110 minutes (55 time units) and 100 minutes (50 time units) respectively.

Class	LPAS_DG	*Gcμ*	FCFS
1	0.27	0.23	0.23
2	0.28	0.23	0.23
3	0.28	0.24	0.23
4	0.25	0.23	0.23
Overall	0.27	0.23	0.23

Table 7.24: Overall results of experiment 9

Discussion

The results of the simulation and our testing environment were similar. The Gcμ policy performed as well as the FCFS policy and slightly better than the LPAS_DG policy.

7.5.2 Experiment 10

This experiment was conducted on Setting *LoLo* with the following parameters:

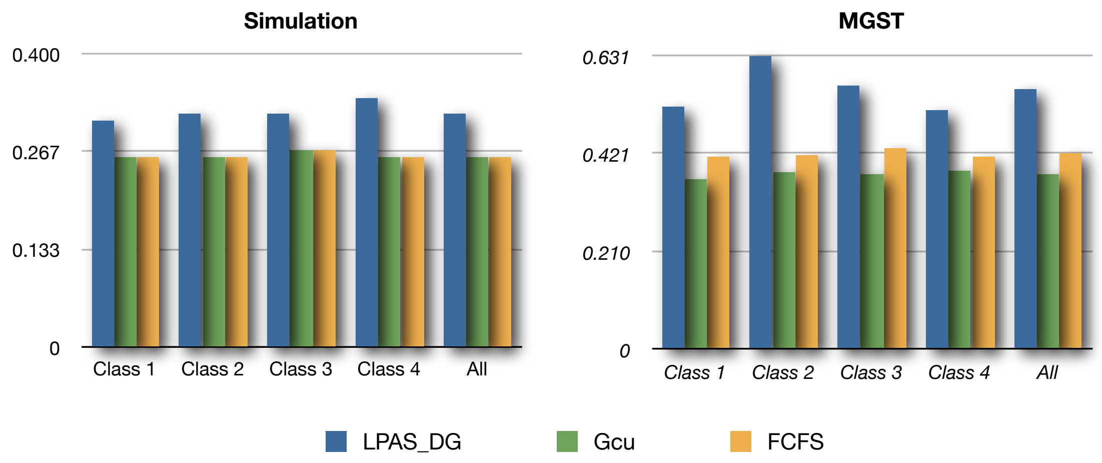


Figure 7.12: Experiment 9 results.

- The availabilities of the machines were as follows:

$$a_j = \begin{cases} 0.50 & \text{if } j = 2, 11 \text{ or } 19 \\ 0.75 & \text{if } j = 3, 12 \text{ or } 20 \\ 1.00 & \text{otherwise} \end{cases}$$

- This experiment included no machine failures.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.25. The metric used here is the mean response time. The confidence level was 95%.

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.28, 0.28)	(0.24, 0.24)	(0.24, 0.24)
2	(0.30, 0.30)	(0.24, 0.24)	(0.24, 0.24)
3	(0.27, 0.27)	(0.25, 0.25)	(0.25, 0.25)
4	(0.32, 0.32)	(0.24, 0.24)	(0.24, 0.24)
Overall	(0.30, 0.30)	(0.24, 0.24)	(0.24, 0.24)

Table 7.25: Results of simulation 10

Our testing environment was used to conduct this experiment, and the results are shown in Table 7.26. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 6%, 10% and 11% respectively of the assumed arrival rates. The LPAS_DG test took 156 minutes (78 time units), while the Gcμ and FCFS tests took 120 minutes (60 time units) and 100 minutes (50 time units) respectively.

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.39	0.32	0.29
2	0.39	0.32	0.29
3	0.35	0.33	0.30
4	0.36	0.33	0.29
Overall	0.37	0.33	0.29

Table 7.26: Overall results of experiment 10

Discussion

The results of the simulation and our testing environment were similar. The Gcμ policy performed as well as the FCFS policy and slightly better than the LPAS_DG policy. The performance of these policies are close to each other. In this setting, the decision of what policy to deploy should be based on other factors.

7.5.3 Experiment 11

This experiment was conducted on Setting *LoLo* with the following parameters:

- All machines were dedicated in this experiment ($a_j = 1.0, \forall j$).
- This experiment included machine failures. The mean uptime was 50 time units and the mean failure period was 2 time units. The periods were exponentially distributed.

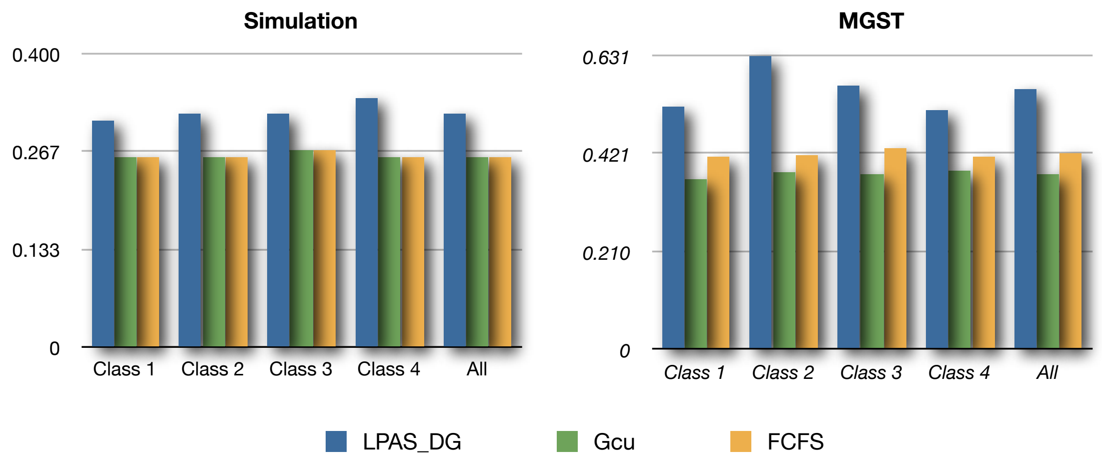


Figure 7.13: Experiment 10 results.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.27. The metric used here is the mean response time. The confidence level was 95%. Our testing environment was used to conduct this

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.25, 0.25)	(0.21, 0.21)	(0.21, 0.21)
2	(0.24, 0.24)	(0.21, 0.21)	(0.21, 0.21)
3	(0.24, 0.24)	(0.21, 0.21)	(0.22, 0.22)
4	(0.24, 0.24)	(0.21, 0.21)	(0.21, 0.21)
Overall	(0.24, 0.24)	(0.21, 0.21)	(0.21, 0.21)

Table 7.27: Results of simulation 11

experiment, and the results are shown in Table 7.28. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 3%, 2% and 5% respectively of the assumed arrival rates. The LPAS_DG test took 50 minutes (100 time unit), while the Gcμ and FCFS tests took 243 minutes (486 time units) and 55 minutes (110 time units) respectively.

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.35	0.26	0.24
2	0.34	0.26	0.24
3	0.33	0.27	0.24
4	0.29	0.26	0.24
Overall	0.33	0.26	0.24

Table 7.28: Overall results of experiment 11

Discussion

The results of the simulation and our testing environment were similar. The Gcμ policy performed as well as the FCFS policy and slightly better than the LPAS_DG policy.

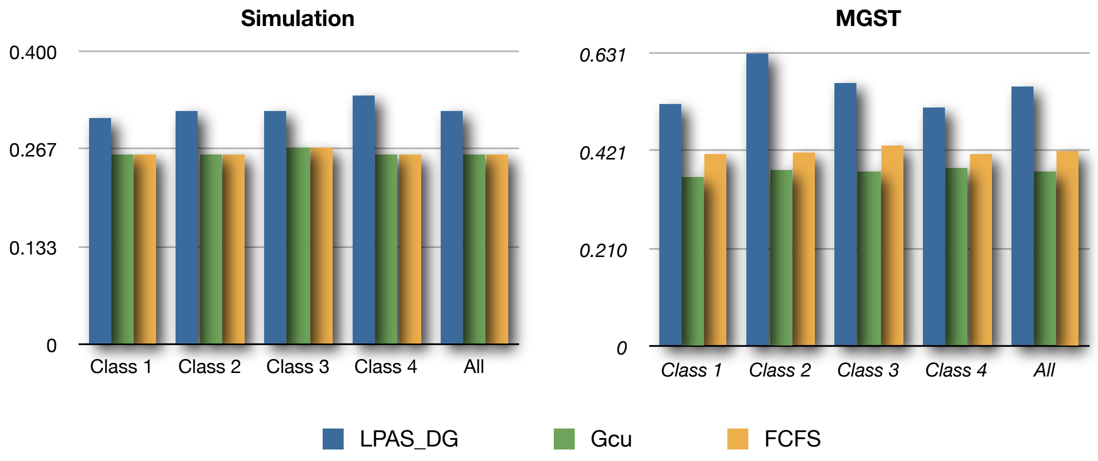


Figure 7.14: Experiment 11 results.

7.5.4 Experiment 12

This experiment was conducted on Setting *LoLo* with the following parameters:

- The availabilities of the machines were as follows:

$$a_j = \begin{cases} 0.50 & \text{if } j = 2,\ 11 \text{ or } 19 \\ 0.75 & \text{if } j = 3,\ 12 \text{ or } 20 \\ 1.00 & \text{otherwise} \end{cases}$$

- This experiment included machine failures. The mean uptime was 50 time units and the mean failure period was 2 time units. The periods were exponentially distributed.

Results

The simulations were done using the simulation software used in [3] and the results obtained are shown in Table 7.29. The metric used here is the mean response time. The confidence level was 95%. Our testing environment was used to conduct this

Class	LPAS_DG	$Gc\mu$	FCFS
1	(0.31, 0.31)	(0.26, 0.26)	(0.26, 0.26)
2	(0.32, 0.32)	(0.25, 0.26)	(0.26, 0.26)
3	(0.32, 0.32)	(0.27, 0.27)	(0.27, 0.27)
4	(0.34, 0.34)	(0.26, 0.26)	(0.26, 0.26)
Overall	(0.32, 0.32)	(0.26, 0.26)	(0.26, 0.26)

Table 7.29: Results of simulation 12

experiment, and the results are shown in Table 7.30. At the time when the LPAS_DG, Gcμ and the FCFS tests were stopped the actual arrival rates of job classes were within 3%, 3% and 4% respectively of the assumed arrival rates. The LPAS_DG test took 176 minutes (88 time units), while the Gcμ and FCFS tests took 108 minutes (54 time units) and 100 minutes (50 time units) respectively.

Discussion

The response times in the results of our experiment were significantly higher than the simulation results. The reason behind this is the high load coupled with failures and

Class	LPAS_DG	$Gc\mu$	FCFS
1	0.52	0.37	0.41
2	0.63	0.38	0.42
3	0.57	0.38	0.43
4	0.52	0.38	0.42
Overall	0.56	0.38	0.42

Table 7.30: Overall results of experiment 12

over estimation of the execution rates (the assumed execution rates were higher than the actual ones in this experiment).

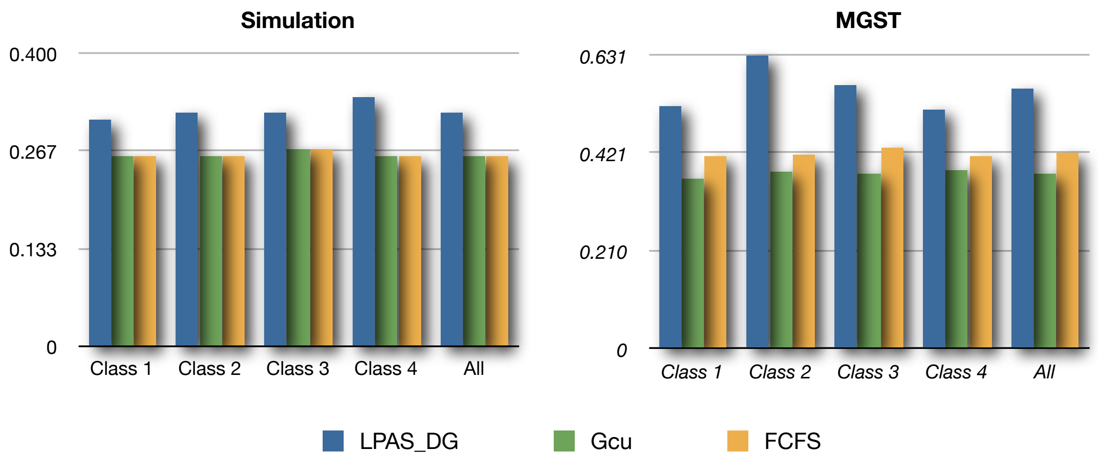


Figure 7.15: Experiment 12 results.

Chapter 8

Conclusion

8.1 Discussion

8.1.1 Testing Environment

We believe that the testing environment developed will prove to be very beneficial for theorists. Not only can the testing environment be used to test scheduling schemes, but the software itself was designed to be extensible in order to include additional features. Having such a testing environment allows researchers to do the following:

- Verify that the scheduling policies designed can be implemented.
- Validate the scheduling policies.
- Verify that the assumptions made actually hold and are reasonable.
- Determine the weak points in a scheduling policy and potentially improve them.

8.1.2 LPAS_DG implementation

Modifications

The LPAS_DG scheduling policy explained in Section 4.4.6 was implemented for the first time in our testing environment. Here we give a few remarks regarding the implementation of this policy.

The LPAS_DG policy is silent on how to choose a server if there is more than one available to serve a job. For simplicity, in our first implementation we chose the FCFS policy to choose servers, but this resulted in performance degradation. The performance is affected because the scheduling process might be blocked when the head of the available servers queue is a server capable of executing a limited number of job classes and none of the currently queued jobs belong to any of these classes. The FCFS implementation was modified to remove the head of the server queue and insert it at the back of the queue, if there are jobs in the jobs queue but this server is not able to execute any of them. However, we believe that the performance of the LPAS_DG can be further improved by employing a suitable policy to choose servers from the available servers queue, especially in the case of a low or medium load on the system. We believe that further research must be conducted to come up with a suitable policy. However, we recommend the LPAS scheduling policy for clusters [4] to be considered as a possible solution, since this policy is suitable for choosing servers for jobs in heterogeneous environments. This modification is not necessary but could improve the performance under low or meduim loads.

The LPAS_DG policy decisions depend on a matrix called δ^* which is produced by solving a linear programming problem (Section 4.4.6). The δ^* matrix depends on the values of a_j. As a result, in [3] it is suggested that a new δ^* matrix must be produced at every availability/unavailability event.

> *Whenever a machine becomes available or unavailable, the scheduler solves the allocation LP to find δ^*.*

Since the matrix δ^* depends on a_j, and the machines' a_j varies between the availability and unavailability events, we think that δ^* should be updated every time any a_j changes. This solution is expensive to implement because it is very hard to notify the mapper of every change to any a_j. In addition, this will require solving the allocation LP frequently, which is also expensive and will raise a scalability problem. To solve this issue, we assumed a time resolution T_{system} (e.g. 10 minutes). The values of a_j are sent to the Mapper periodically which causes it to solve the allocation LP once again after receiving the values of a_j. The determination of an optimal time resolution length is open to research. We believe that this modification is necessary to make LPAS_DG scalable.

Robust Modifications

In some experiments the performance of the scheduling schemes differed from the simulation results due to the machines experiencing an overload. This happens when machines are highly loaded (at least 80%). The different sources of errors that can occur in a real system can significantly raise the load, even potentially causing instability in the system. These errors can be caused by:

1. **The actual arrival rate being larger than the assumed one.** This results in receiving more jobs than expected and increasing the load on the machines.

2. **Overestimation of processing rates.** This results in executing the jobs in more time than expected causing the server to be busier thus the load increases on the servers.

3. **Overhead caused by communication and scheduling delays.** Assume that a server announces its availability at time t_1, then the mapper learns of the availability of this server at time t_2 and consequently performs the scheduling and chooses a job at time t_3 and then sends the job. The server then receives the job and starts the execution at time t_4. At time t_5 the server finishes the job execution but only at time t_6 does the mapper learn that the job is done, obtaining the results at t_7. In the model, the processing time is considered to be $t_6 - t_5$, but in the actual implementation, there is an overhead of $(t_5 - t_1) + (t_7 - t_6)$. This overhead is usually negligible, but sometimes it affects the load on the system, especially if $t_6 - t_5$ is small compared to the overhead.

4. **Machine failures.** Although machines failure can be incorporated in the workload models, they can still increase the effective load due to the fact that it takes

time for the mapper to realize that a server is down. This time is wasted and effectively increases the load. For example, when using LPAS_DG, suppose that server 3 is the only server executing jobs from class 1, and the execution time is 5 minutes. If server 3 fails when executing a particular job and the "time-out" parameter was set to 3 times (i.e. 3 times the estimated execution time should elapse before considering the job "timed out"), then the Mapper will not consider server 3 down until 15 minutes have elapsed from the moment that the job was sent. These 15 minutes were essentially lost, with arriving jobs from class 1 accumulating in the queue at the Mapper within that time.

If any or all of the above factors cause a significant increase in the load, the performance of the scheduling scheme will deteriorate.

The LPAS_DG policy suffered the most in our experiments from the above factors due to the aggressive nature of this policy in minimizing the number of machines to execute each job class. Another factor is the exclusivity that can happen when using this policy. When one class can be executed by a small number of machines, then the performance depends only on these machines, so the effect of the factors mention above is magnified. Contrast this with FCFS, where if a machine under performs, the effect is less obvious since this under performing machine can get help from other (potentially over performing) machines. Finally, the scheduling delay can contribute to the time needed to process jobs, effectively raising the load on machines for all policies. The scheduling delay for LPAS_DG is slightly larger than G$c\mu$ due to the overhead of solving the LP, while both policies have a larger delay than the FCFS policy due to the delay that occurs when choosing a job, as the LPAS_DG or G$c\mu$ policies must check multiple queues to choose the suitable job where the FCFS policy has only one queue. The mentioned issues (exclusivity and scheduling delay) cause the LPAS_DG policy to be the most sensitive to the above four factors, the G$c\mu$ policy to be the next most sensitive and then the FCFS policy (least scheduling delay) is the least sensitive.

After discussing the reasons that can effect the robustness of the LPAS_DG, we provide the following suggestions to improve robustness:

1. **Arrival rates estimation improvement.** Since the LPAS_DG scheme depends on solving the LP problem and that in turn depends on values that include arrival rates of job classes, estimates should be as close as possible. To do so, we propose that the actual arrival rates should be monitored (a feature that our tool provides), and check the values against the estimated values every specific time ($T_{arrival\_rate}$) and resolve the LP if one of the actual values differs from the estimated one by a specific threshold percentage ($Th_{arrival\_rate}$) that depends on the load and the job class. $T_{arrival\_rate}$ could be a specific time period or a number of job arrivals from a class (e.g. 10 jobs). We believe that this solution is not computationally costly, since the checking operation requires $O(N)$ time and $O(1)$ space. We expect the number of jobs classes to be relatively small, so there should be no scaling issues.

2. **Avoiding processing rates underestimation.** We propose that every processing rate entry (for a specific server for a specific job class) is modified then checked (against the estimated peer) whenever a job is done, then the LP is resolved if that entry differs from the estimated one by a specific threshold percentage ($T_{processing\_rate}$) that depends on the load and the job class. This solution requires$O(NM)$ space and $O(1)$ time.

3. **Lessen the affect of communication and scheduling delays.** Let $p_{i,j}$ be an estimation of the value
$$\frac{1/\mu_{i,j}}{1/\mu_{i,j} + \tau_j} \tag{8.1}$$
where τ is the communication and scheduling delay for machine j.

   In the example mentioned in factor 3 on p.93 p would be
$$\frac{t_6 - t_5}{t_7 - t_1} \tag{8.2}$$

   We propose that all execution rates must be multiplied by p before resolving the LP to take this effect into consideration.

4. **Lessen the machine failure affect.** We propose choosing a low value for the time out, which will result in allowing the mapper to faster indicate server failures. The downside of this approach is that the mapper might consider a server failed one when it is not.

To sum up, we believe that some modifications to the LPAS_DG policy should be performed to make it more implementable, some of which have already been done in our implementation. We believe that these changes will make the LPAS_DG an excellent (possibly the best) solution in heterogeneous environments and a good one in other environments.

8.2 Future Work

The following areas and software additions are of interest for future work:

- The implementation of more scheduling polices and the conducting of experiments, in addition to the implementation of more probability distributions. Implementing scheduling schemes and testing them is the reason why this software was built.
- Changing the software layer used. This can be useful to allow testers to use Windows or Linux computers as servers. This can be done by extending the *Executer* abstract class and implementing its methods properly. This might allow the testing environment to expand and allow testers to ask users at home to install the Puller module on their machines and therefore allowing experiments with a larger number of machines.
- Adding the feature for reading real workload traces and simulating them.
- Launching an open source project to maintain the software and expand it. We believe that release of the source code and putting the software in the open source domain will result in the expansion of this tool. Other research can help develop this testing environment and use it. However, we recommend that

the open source project should be supervised by a committee to guarantee the correctness of the software updates.

- Finding a suitable policy to choose a server among a set of servers when LPAS_DG is used. We believe that this will improve the LPAS_DG performance especially in the case where the system is not highly loaded.
- Finding optimal values of parameters mentioned in Section 8.1.2 . The T_{system} value for example, should lessen the communication in the grid system while aiding the LPAS_DG policy in producing an updated δ^* that will ultimately maximize the performance.

Bibliography

[1] ABC@home "http://abcathome.com/", last visited March 10, 2008.

[2] Artificial Intelligence System "http://www.intelligencerealm.com/aisystem/system.php", last visited March 10, 2008.

[3] I. Al-Azzoni and D. Down, "Dynamic scheduling for heterogeneous desktop grids," in *Grid 2008*, 2008.

[4] I. Al-Azzoni and D. Down, "Linear programming based affinity scheduling of independent tasks on heterogeneous computing systems," in *IEEE Transactions on Parallel and Distributed Systems, to appear*, 2008.

[5] S. Andradóttir, H. Ayhan, and D. G. Down, "Compensating for failures with flexible servers," *Operations Research*, vol. 55, no. 4, pp. 753–768, 2007.

[6] S. Andradóttir, H. Ayhan, and D. G. Down, "Dynamic server allocation for queueing networks with flexible servers," *Operations Research*, vol. 51, no. 6, pp. 952–968, 2003.

[7] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for bag-of-tasks applications on desktop grids," in *7th IEEE/ACM International Conference on Grid Computing*, pp. 56–63, 2006.

[8] Apple, "Mac OS X Server, Xgrid administration," tech. rep., Apple Inc, 2005.

[9] APS@home "http://www.apsathome.org/", last visited March 10, 2008.

[10] R. K. Armstrong, "Investigation of Effect of Different Run-Time Distributions on Smartnet Performance," Master's thesis, Naval Postgraduate School, September 1997.

[11] A. L. Beberg and V. S. Pande, "Storage@home: Petascale distributed storage," in *Proceedings of the Conference of 21th International Parallel and Distributed Processing Symposium*, pp. 1–6, 2007.

[12] BURP Web site "http://burp.boinc.dk/", last visited March 10, 2008.

[13] S. Choi, H. Kim, E. Byun, M. Baik, S. Kim, C. Y. Park, and C.-S. Hwang, "Characterizing and classifying desktop grid," in *Seventh IEEE International Symposium on Cluster Computing and the Grid*, pp. 743–748, 2007.

[14] S. Choi, H. Kim, E. Byun, and C. Hwang, "A taxonomy of desktop grid systems focusing on scheduling," Tech. Rep. KU-CSE-2006-1120-01, Department of Computer Science and Engineering, Korea University, November 2006.

[15] Compute Against Cancer "http://www.computeagainstcancer.org/", last visited March 10, 2008.

[16] Condor Home Page "http://www.cs.wisc.edu/condor/ ", last visited March 20, 2008.

[17] Distributed.net "http://www.distributed.net/", last visited March 8, 2008.

[18] Distributed.net "http://www.distributed.net/ogr", last visited March 8, 2008.

[19] P. Domingues and A. A. L. Silva, "Scheduling for fast turnaround time on institutional desktop grid," tech. rep., CoreGRID, January 2006.

[20] FightAIDS@home "http://fightaidsathome.scripps.edu/", last visited March 10, 2008.

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

[22] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," tech. rep., Argonne National Laboratory, University of Chicago, University of Southern California IBM Corporation, 2002.

[23] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with smartnet," in *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, p. 3, IEEE Computer Society, 1998.

[24] World Community Grid and the Oswaldo Cruz Institute - Fiocruz - Genome Comparison Project "http://www.dbbm.fiocruz.br/GenomeComparison", last visited March 9, 2008.

[25] J. Gosling and H. McGilton, "The Java language environment," tech. rep., Sun Microsystems, May 1996.

[26] Y.-T. He, I. Al-Azzoni, and D. Down, "MARO - MinDrift affinity routing for resource management in heterogeneous computing systems," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 71–85, 2007.

[27] World Internet Usage Statistics News and Population Stats. "http://www. internetworldstats.com/", last visited February 29, 2008.

[28] Internet Systems Consortium. "http://www.ISC.org/", last visited February 25, 2008.

[29] Javadoc Tool Home Page "http://java.sun.com/j2se/javadoc/", last visited February 25, 2008.

[30] D. G. John L. Hennessy, David A. Patterson and K. Asanovic, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2003.

[31] R. B. Klaus Krauter and M. Maheswaran, “A taxonomy and survey of grid resource managment systems,” *Software: Practice and Experience*, vol. 32, no. 2, pp. 135–164, 2002.

[32] D. Kondo, A. A. Chien, and H. Casanova, “Resource management for rapid application turnaround on enterprise desktop grids,” in *Proceedings of the ACM/IEEE Conference on Supercomputing*, p. 17, 2004.

[33] E. Kourpes, “Grid computing: Past, present and future, an innovation perspective,” tech. rep., IBM Corporation, June 2006.

[34] A. Mandelbaum and A. L. Stolyar, “Scheduling flexible servers with convex delay costs: Heavy-traffic optimality of the generalized cμ-rule,” *Operations Research.*, vol. 52, no. 6, pp. 836–855, 2004.

[35] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[36] Muthucumaru, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, “Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems,” in *Proceedings of the 8th Heterogeneous Computing Workshop*, pp. 30–44, 1999.

[37] PiHex - a distributed effort to calculate Pi “http://oldweb.cecm.sfu.ca/projects/pihex/”, last visited March 9, 2008.

[38] Proteins@home “http://biology.polytechnique.fr/proteinsathome”, last visited March 10, 2008.

[39] X. Ren, R. Eigenmann, and S. Bagchi, “Failure-aware checkpointing in fine-grained cycle sharing systems,” in *HPDC ’07: Proceedings of the 16th international symposium on High performance distributed computing*, pp. 33–42, ACM, 2007.

[40] Rosetta@home "http://boinc.bakerlab.org/rosetta/", last visited March 10, 2008.

[41] Seti@home "http://setiathome.berkeley.edu/", last visited March 8, 2008.

[42] SHA-1 Collision Search "http://boinc.iaik.tugraz.at/", last visited March 10, 2008.

[43] R. Shah, B. Veeravalli, and M. Misra, "On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 12, pp. 1675–1686, 2007.

[44] Platform Computing "http://www.platform.com ", last visited March 20, 2008.

[45] Sodoku Project Web site "http://dist2.ist.tugraz.at/sudoku/", last visited March 10, 2008.

[46] Spinhenge@home "http://spin.fh-bielefeld.de/", last visited March 10, 2008.

[47] Java AWT: Delegation Event Model " http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/event
last visited June 1, 2008.

[48] uFluids Web Site "http://www.ufluids.net/", last visited March 10, 2008.

[49] Xgrid@Stanford "http://cmgm.stanford.edu/ cparnot/xgrid-stanford/", last visited February 25, 2008.

[50] N. R. Yongnan Ji, Jin Hao and A. Lendasse, "Direct and recursive prediction of time series using mutual information selection," in *Computational Intelligence and Bioinspired Systems*, pp. 1010–1017, Springer Berlin / Heidelberg, 2005.

Appendix A

Source Code and Javadoc Documentation CD

The accompanying Compact Disc contains the source code of the testing environment software and its Javadoc Documentation.

Appendix B

User Manual

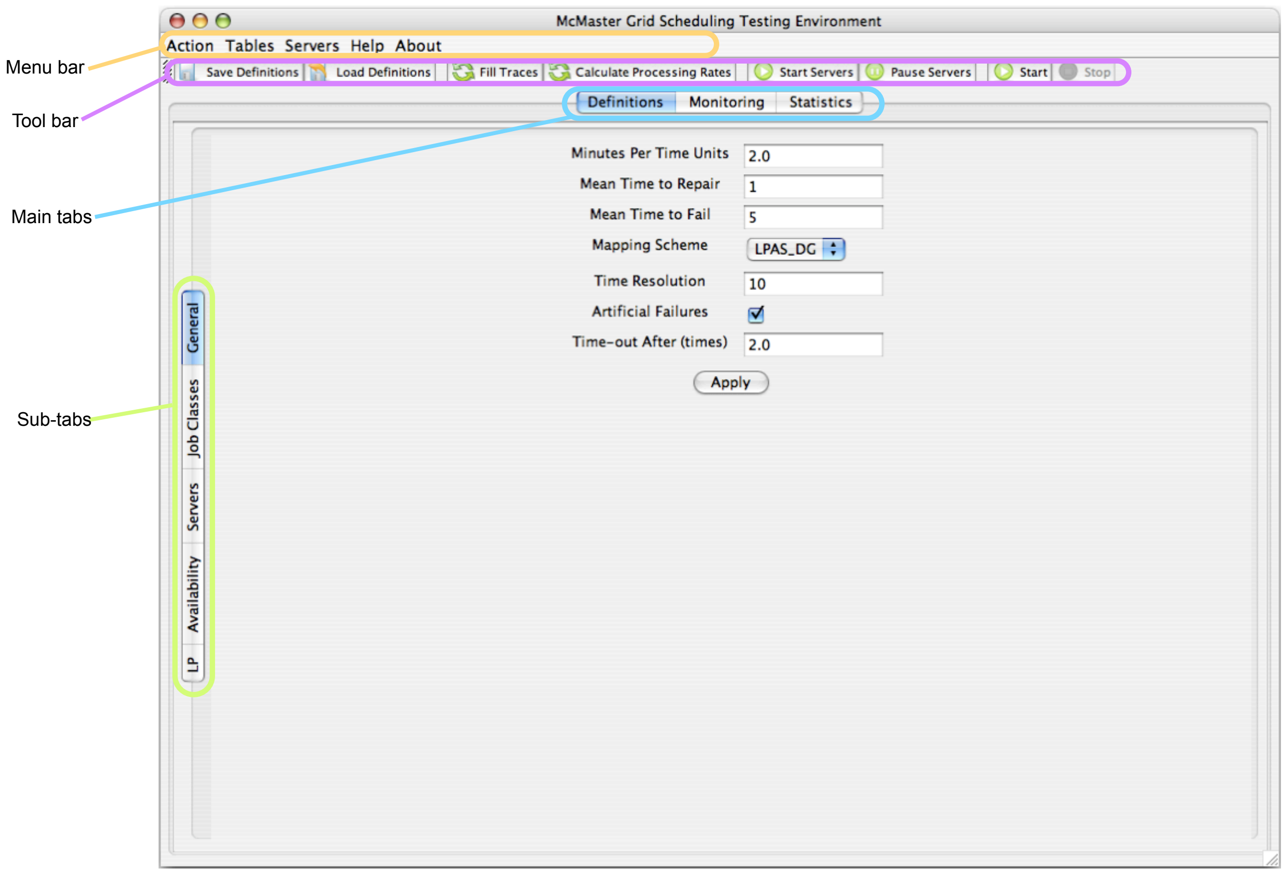
This appendix serves as a user manual for the developed testing environment. In order to conduct an experiment, the tester should prepare the machines which will serve as servers. After that the tester has to define the parameters of the system, then start the test. The tester can then monitor the test and finally read and store statistics about the test. The remainder of this chapter discusses these phases.

B.1 Preparation of Servers

In this phase the execution layer should be prepared. Two steps should be taken at every server in order to use the Xgrid execution layer.

- The Xgrid controller and agent services should be turned on. See Appendix C for details.

- The *Puller.jar* executable should be running, by executing the following command: *java -jar Puller.jar*. The *Puller.jar* file can be found on the Compact Disc of Appendix A. This executable contains the Puller module of the software.



Minutes Per Time Units	2.0
Mean Time to Repair	1
Mean Time to Fail	5
Mapping Scheme	LPAS_DG
Time Resolution	10
Artificial Failures	✓
Time-out After (times)	2.0

Figure B.1: General Screen Shot

B.2 User Interface

The User interface of the software is divided into the tool bar (where the most used actions have short cuts), the menu bar (where system functions can be invoked) and the main tabs. Each main tab is responsible for one phase of the test or a particular functionality and has several sub tabs. The remaining sections discuss these tabs in detail.

B.3 Definition Phase

This phase is done through the main tab labelled *Definitions*. In this phase the tester should define the parameters of the system, including:

- General Parameters (e.g time units in minutes or scheduling policy to be used)
- Job classes
- Servers
- Server Availability

B.3.1 General Parameters

These general parameters are accessed under the main tab *Definitions* and the sub tab *General*.

- Time Unit in Minutes: this parameter defines the length of the time unit [1]used in a test in minutes.
- Mean Time to Repair (MTTR): this parameter defines the mean length of the failure periods for all the servers when the artificial failures option is enabled.
- Mean Time to Failure: this parameter defines the mean length of the up-time periods for all the servers when the artificial failures option is enabled.
- Mapping Scheme: This parameter determines the scheduling policy used in a test.
- Time Resolution: This parameter is T_{system}. Please refer to Section 8.1.2.
- Artificial Failures and T/O (Time-out): Artificial failures can be simulated to study the effect of failures. This parameter determines whether the artificial failures option is enabled or not.

[1]A time unit is a hypothetical time unit used as the unit of all time quantities in the system. (e.g. the units of execution rates and arrival rate is task per time unit)

- Time-out: Every job has an estimated execution time. If a server failure occurs while executing a job, the completion notification will not reach the mapper. The mapper waits for that job to be completed for n times the expected execution time, where n is the "Time-out after" parameter. Then the mapper invokes the *handleTimeOut* method of the active scheduling scheme.

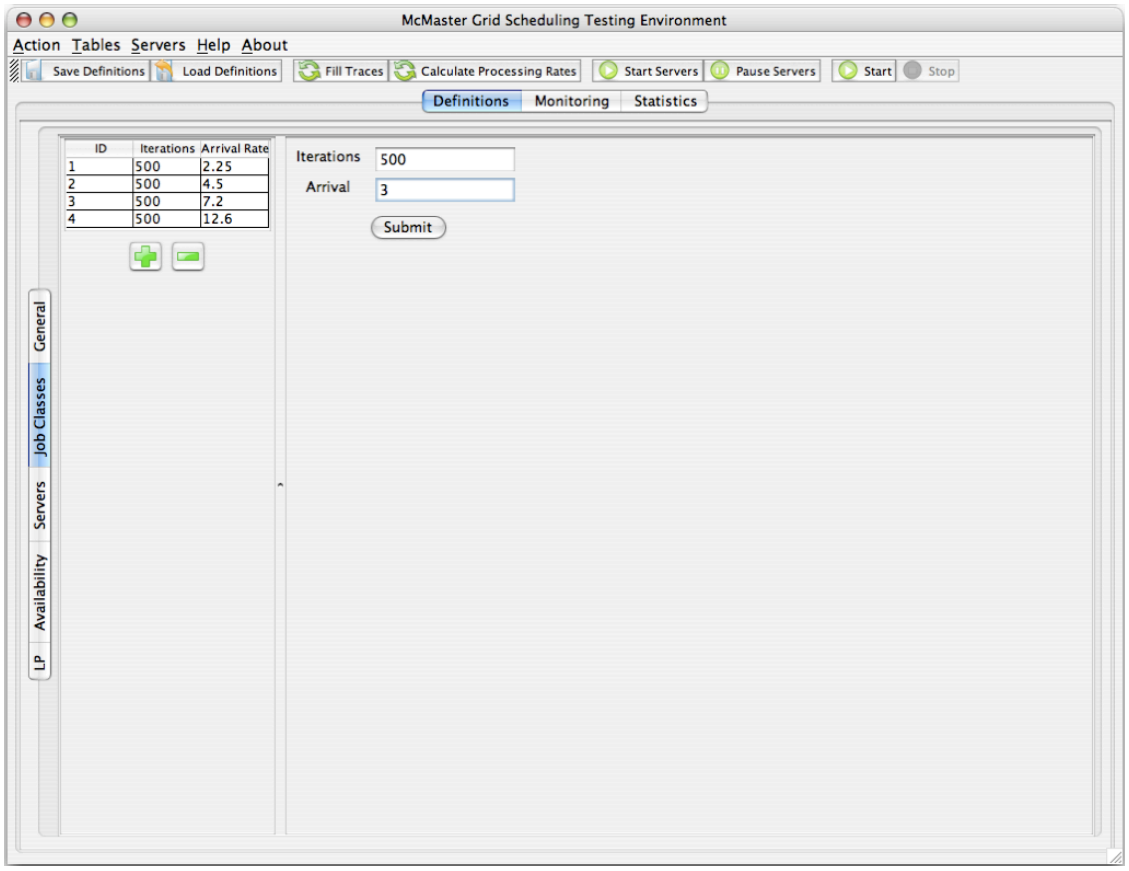
After the user has set all of the parameters, she should click on the Apply button (Figure B.1).

B.3.2 Job Classes

The job classes are defined in this phase. Every job class has an ID, iterations and arrival rate. The ID of a job class the number of the column which represents this job class in the μ matrix (Section 3.1). The iterations of a job class is the mean number which the jobs of this class have as iterations (Appendix D). In other words, the number of iterations is the number of times the triple loop in Appendix D is executed. The iterations number will affect the real execution rates of the machines. The arrival rate is the mean number of jobs that will arrive to the system per time unit under an exponential interarrival time distribution. To add a job, click on the plus button and fill the iterations and the arrival rate. Then click on the Submit button (Figure B.2). The classes will be added in order. To delete a job, select it and then click the minus button.

B.3.3 Servers

The servers are defined in this phase. There are many ways that this can be done. Obviously, the servers to be defined should be the ones set up in the Preparation Of Servers phase (B.1). One way of adding servers is to click the plus button and insert the information related to the server (Figure B.3). The full canonical hostname should be inserted as the hostname. The password of the Xgrid layer should be entered as the password. The Iterations and the Processing Time are important parameters. Each server should be sent the loop job (Appendix D) to execute a few times, then



ID	Iterations	Arrival Rate
1	500	2.25
2	500	4.5
3	500	7.2
4	500	12.6

Iterations: 500

Arrival: 3

Figure B.2: Job Classes Screen Sub Tab

the time required by this server to execute each loop is measured, and the average is taken. For example if the machine itb237-01.cas.mcmaster.ca is to be added to the set of servers, the software must know how long it takes this machine to execute the loop job for a particular number of iterations. This allows the software to predict the actual processing rates for machines. The iterations and time to process in minutes are inserted when a server is desired to be added. The servers will be added in order. The ID of a server is its order in the μ matrix (Section 3.1).

For convenience, a file can be prepared where each line corresponds to one server. The file extension should be *srs*. Each line should be structured in the following format (spaces are ignored):

hostname ; password ; time to process (minutes) ; iterations

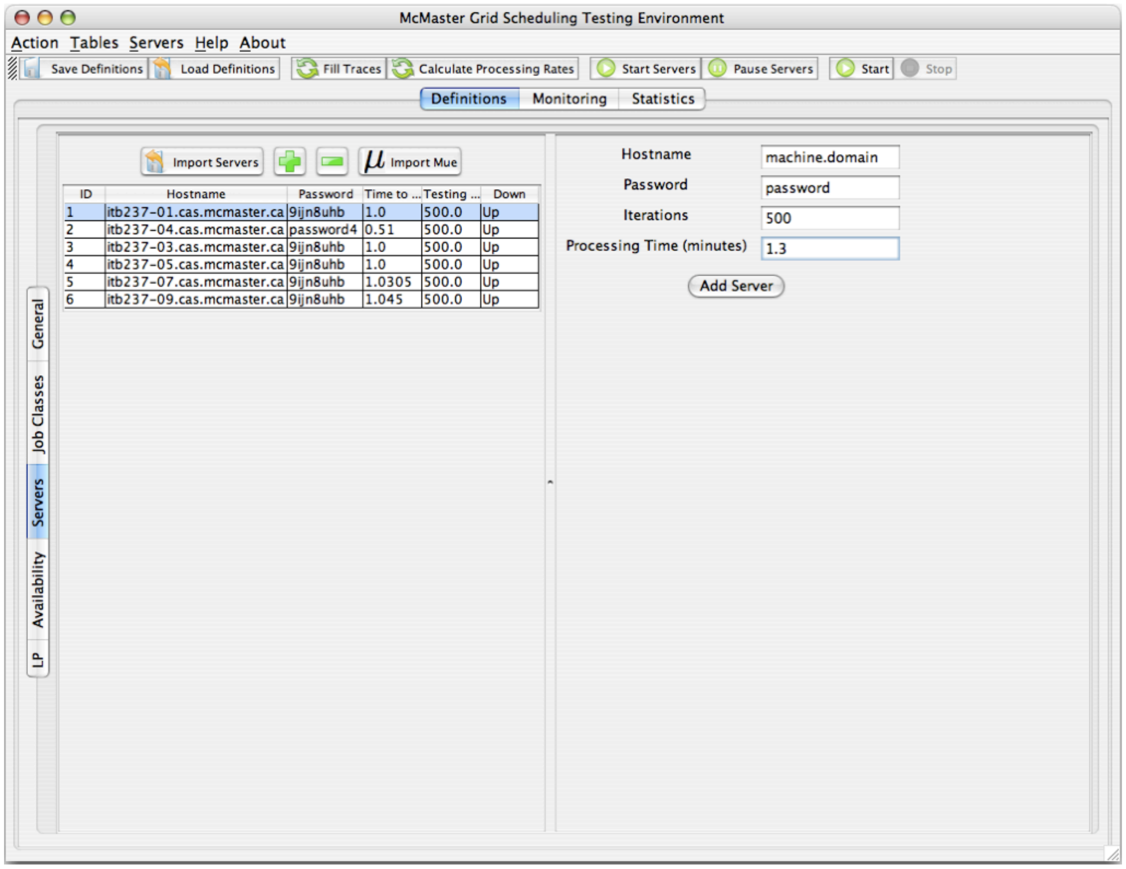
For example the following are the contents of the file imported before the screen shot in Figure B.3 was taken.

```
itb237-01.cas.mcmaster.ca; 9ijn8uhb; 1 ; 500
itb237-04.cas.mcmaster.ca;password4; 0.51 ; 500
itb237-03.cas.mcmaster.ca;9ijn8uhb; 1 ; 500
itb237-05.cas.mcmaster.ca;9ijn8uhb; 1 ; 500
itb237-07.cas.mcmaster.ca;9ijn8uhb; 1.0305 ; 500
itb237-09.cas.mcmaster.ca;9ijn8uhb; 1.045 ; 500
```

To delete a server, it should be selected and then the minus button should be clicked.

Processing Rates

To modify or view the processing rates of a server, the user should select the server by clicking on it. The *Processing Rates* tab will appear on the right side (Figure B.4). The real rates (second column) are those which the software estimated using the iterations and the time to process values inserted by the tester. The assumed rates (third column) can be changed. It is recommended that the tester does not force the server to be more than 4 times faster than the real rates (e.g. if the real rate is 4.0, it is recommended that the assumed rate is not larger than 16.0). Basically, the *Assumed Rate* column for a server with ID i is the i^{th} column in the μ matrix. To change an assumed rate, the user has to click on the appropriate cell and type a new number then press enter. For convenience, the tester can import all of the processing rates of a setting using the *Import PR* button. The file should be a text file with extension *mue*. The file format should be similar to the μ matrix but entries are separated by commas. The following is the content of the *mue* file used in the *LoHi* experiments (Chapter 7).



ID	Hostname	Password	Time to ...	Testing ...	Down
1	itb237-01.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
2	itb237-04.cas.mcmaster.ca	password4	0.51	500.0	Up
3	itb237-03.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
4	itb237-05.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
5	itb237-07.cas.mcmaster.ca	9ijn8uhb	1.0305	500.0	Up
6	itb237-09.cas.mcmaster.ca	9ijn8uhb	1.045	500.0	Up

Hostname: machine.domain
Password: password
Iterations: 500
Processing Time (minutes): 1.3

Figure B.3: Servers Screen Sub Tab

2.2, 7, 10.25, 1, 5.7, 0.5, 12, 2.2, 7, 10.25, 1, 5.7, 0.5, 12, 2.2, 7, 10.25, 1, 5.7, 0.5, 12

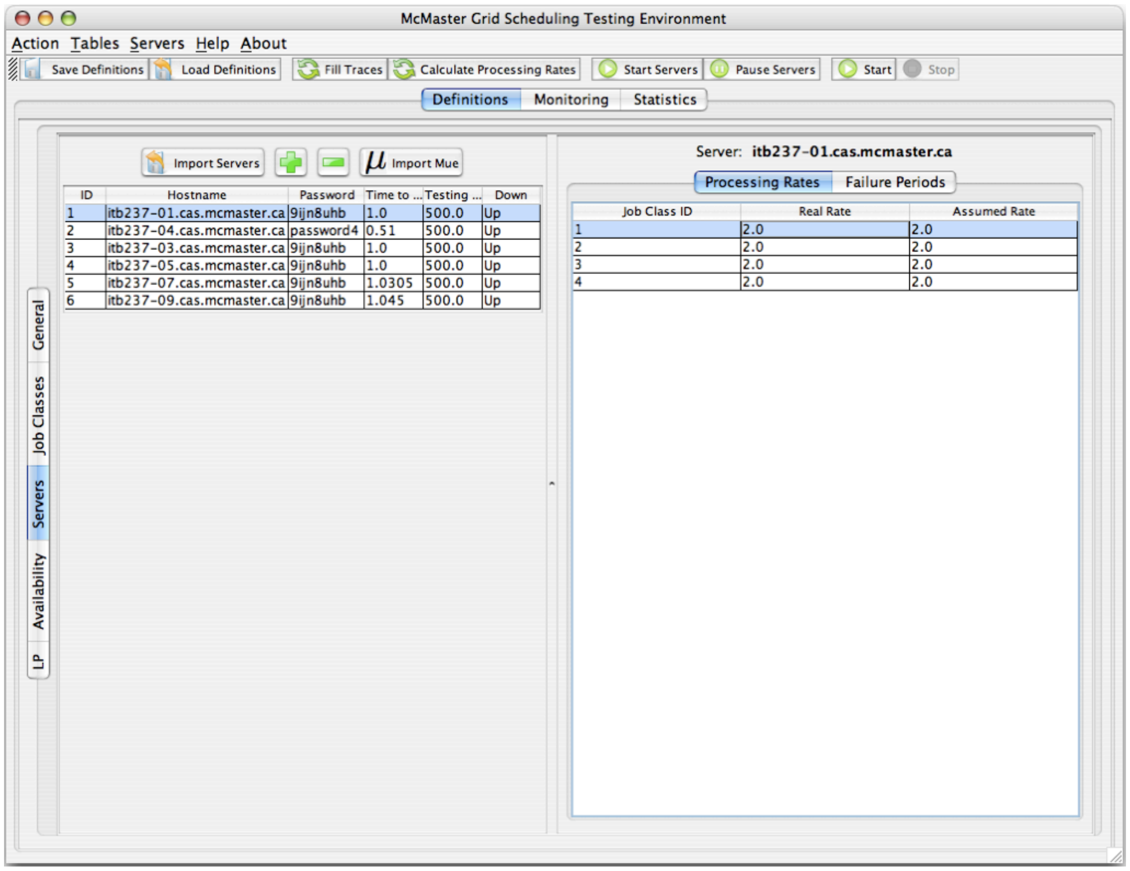
1.95, 7.05, 9.78, 0.95, 5.65, 0.56, 11.85,1.95,7.05,9.78, 0.95,5.65,0.56,11.85,1.95,7.05,9.78, 0.95, 5.65,0.56,11.85

2, 7.25, 10.02, 0.98, 5.75, 0.67, 11.8, 2, 7.25, 10.02, 0.98, 5.75, 0.67, 11.8, 2, 7.25, 10.02, 0.98, 5.75, 0.67, 11.8

2.05, 6.75, 9.99, 1.02, 5.82, 0.49,12.05, 2.05,6.75,9.99,1.02,5.82,0.49,12.05, 2.05, 6.75,9.99,1.02, 5.82, 0.49,12.05

Failure Periods

In the case that the artificial failure option is enabled, the artificial failure of a server can be viewed by clicking on a server, and then selecting the *Failure Periods* sub tab (Figure B.5). To generate new failure traces for all of the servers, the button *Fill*



ID	Hostname	Password	Time to ...	Testing ...	Down
1	itb237-01.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
2	itb237-04.cas.mcmaster.ca	password4	0.51	500.0	Up
3	itb237-03.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
4	itb237-05.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
5	itb237-07.cas.mcmaster.ca	9ijn8uhb	1.0305	500.0	Up
6	itb237-09.cas.mcmaster.ca	9ijn8uhb	1.045	500.0	Up

Server: **itb237-01.cas.mcmaster.ca**

Job Class ID	Real Rate	Assumed Rate
1	2.0	2.0
2	2.0	2.0
3	2.0	2.0
4	2.0	2.0

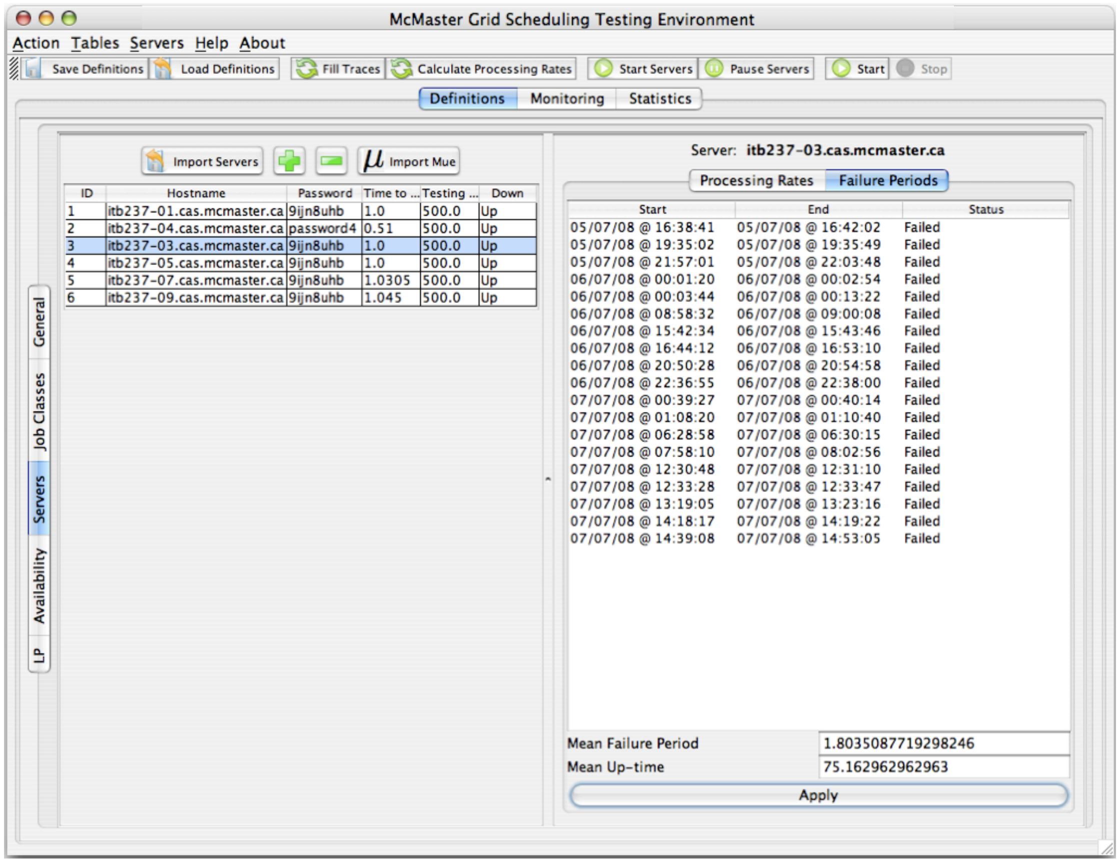
Figure B.4: Servers with processing rates.

Traces should be clicked. The actual mean up-time and the mean failure period of a server are viewed at the bottom. To change these values for each server individually, the user has to change values in the text fields and click on *Apply*.

B.3.4 Availability

Every machine has a puller module running on it. To set up the module a message has to be sent to it. In this phase the messages (and hence the settings) of the puller modules (i.e servers) are prepared and sent.

One or more servers are selected from the table on the left (Figure B.6), then the properties are set in the right side. The properties are:



ID	Hostname	Password	Time to ...	Testing ...	Down
1	itb237-01.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
2	itb237-04.cas.mcmaster.ca	password4	0.51	500.0	Up
3	itb237-03.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
4	itb237-05.cas.mcmaster.ca	9ijn8uhb	1.0	500.0	Up
5	itb237-07.cas.mcmaster.ca	9ijn8uhb	1.0305	500.0	Up
6	itb237-09.cas.mcmaster.ca	9ijn8uhb	1.045	500.0	Up

Server: **itb237-03.cas.mcmaster.ca**

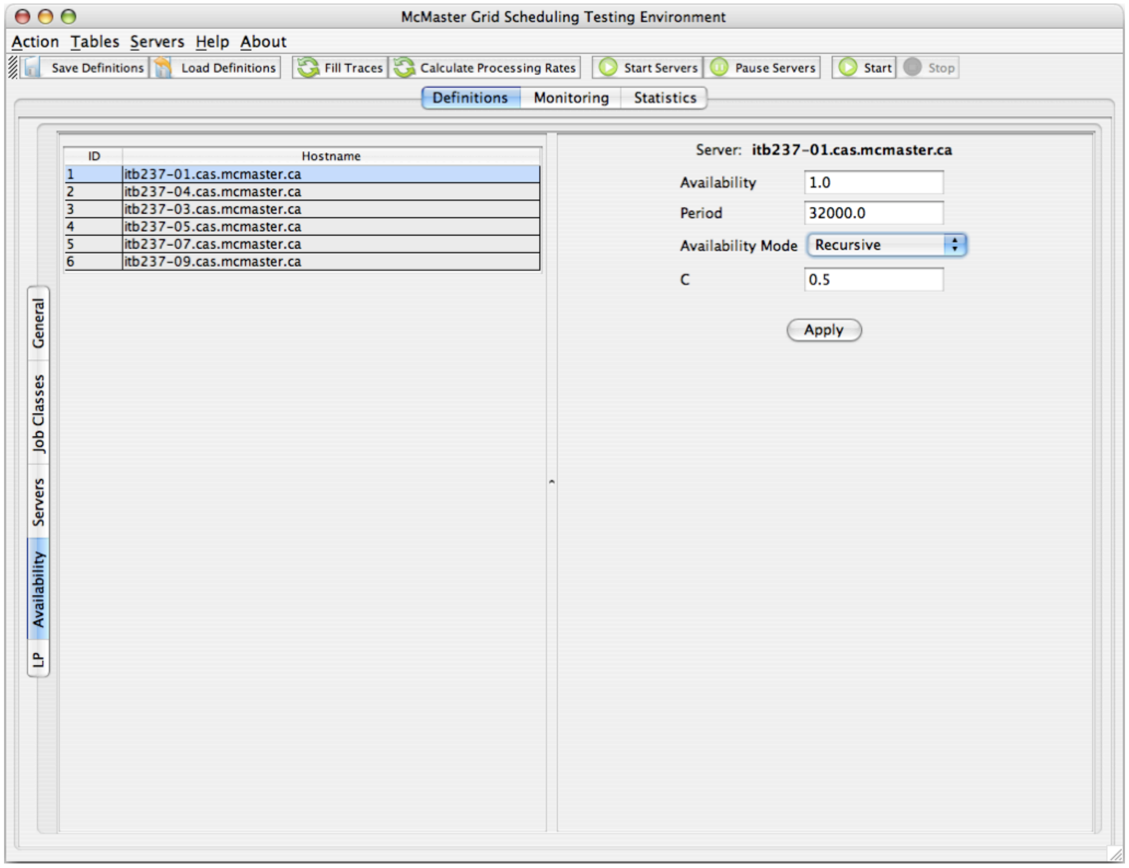
Start	End	Status
05/07/08 @ 16:38:41	05/07/08 @ 16:42:02	Failed
05/07/08 @ 19:35:02	05/07/08 @ 19:35:49	Failed
05/07/08 @ 21:57:01	05/07/08 @ 22:03:48	Failed
06/07/08 @ 00:01:20	06/07/08 @ 00:02:54	Failed
06/07/08 @ 00:03:44	06/07/08 @ 00:13:22	Failed
06/07/08 @ 08:58:32	06/07/08 @ 09:00:08	Failed
06/07/08 @ 15:42:34	06/07/08 @ 15:43:46	Failed
06/07/08 @ 16:44:12	06/07/08 @ 16:53:10	Failed
06/07/08 @ 20:50:28	06/07/08 @ 20:54:58	Failed
06/07/08 @ 22:36:55	06/07/08 @ 22:38:00	Failed
07/07/08 @ 00:39:27	07/07/08 @ 00:40:14	Failed
07/07/08 @ 01:08:20	07/07/08 @ 01:10:40	Failed
07/07/08 @ 06:28:58	07/07/08 @ 06:30:15	Failed
07/07/08 @ 07:58:10	07/07/08 @ 08:02:56	Failed
07/07/08 @ 12:30:48	07/07/08 @ 12:31:10	Failed
07/07/08 @ 12:33:28	07/07/08 @ 12:33:47	Failed
07/07/08 @ 13:19:05	07/07/08 @ 13:23:16	Failed
07/07/08 @ 14:18:17	07/07/08 @ 14:19:22	Failed
07/07/08 @ 14:39:08	07/07/08 @ 14:53:05	Failed

Mean Failure Period	1.8035087719298246
Mean Up-time	75.162962962963

Figure B.5: Servers with failure periods.

- Availability is the a_j value (Section 3.2).
- Period is the time in minutes that a_j will be imposed on the selected servers. The Period should be longer than the time the tests are to be run for.
- Availability Mode is what method of availability prediction is used (Section 3.2). There are three different modes. Choosing different modes will be followed by the inserting of parameters related to that mode.

After preparing the messages, they can be sent to the servers using the *Servers* menu in the menu bar or the *Start Servers* button in the toolbar. In addition, the servers can be paused, pinged or killed. All of these actions can be found under the *Servers* menu in the menu bar. After pausing a server it must be started again to function properly. Pinging can be used to make sure the server is turned on. After killing the server, the puller executable must be run on that server (using $java - jarpuller.jar$) to restart it, as the kill signal makes the puller.jar process exit.



ID	Hostname
1	itb237-01.cas.mcmaster.ca
2	itb237-04.cas.mcmaster.ca
3	itb237-03.cas.mcmaster.ca
4	itb237-05.cas.mcmaster.ca
5	itb237-07.cas.mcmaster.ca
6	itb237-09.cas.mcmaster.ca

Server: **itb237-01.cas.mcmaster.ca**

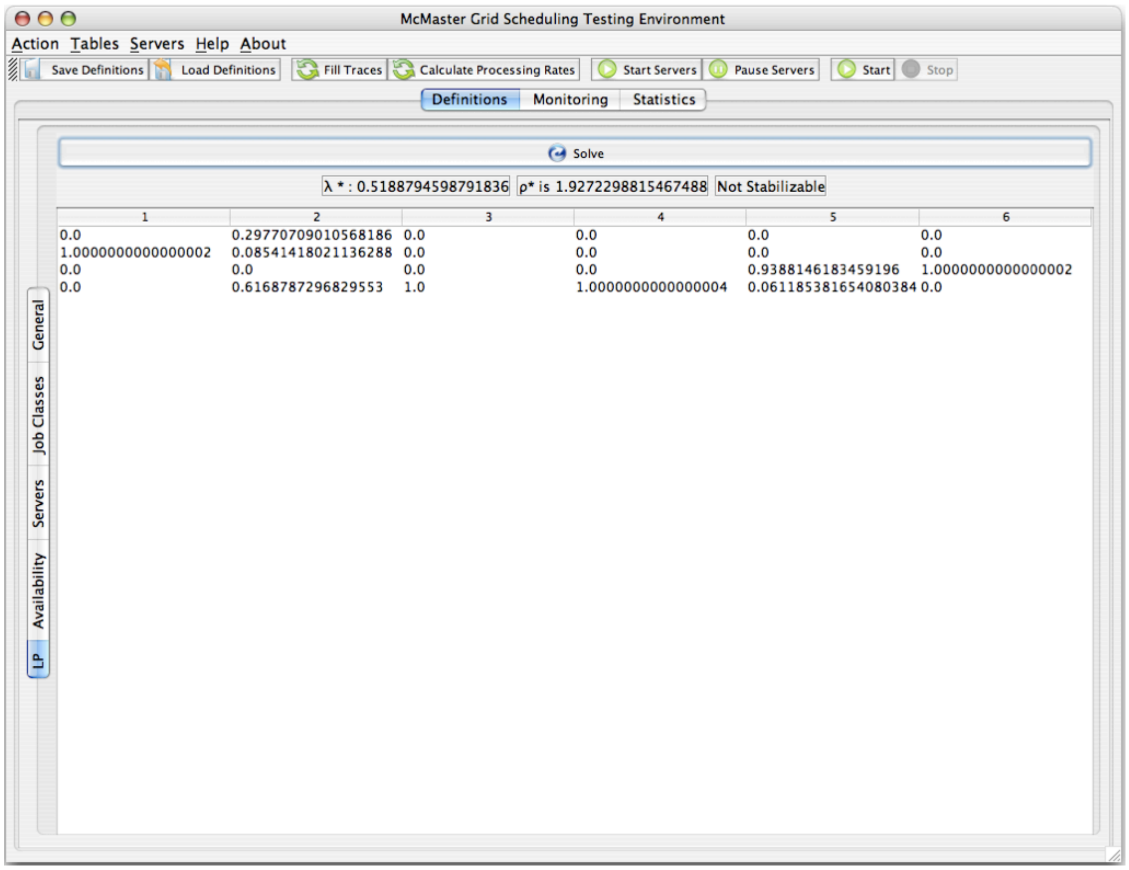
Availability	1.0
Period	32000.0
Availability Mode	Recursive
C	0.5

Figure B.6: Availability Screen Sub Tab

B.3.5 LP

In this phase the LP allocation can be solved. To solve the LP allocation, the solve button should be clicked. The δ^* matrix is then displayed. Also, λ^* and ρ^* are shown as in Figure B.7 (Section 4.4.6).

After the completion of the definitions, you can save them to a file using the *Save definitions* button. Saved definitions can be restored using the *Load definitions* button. All the information in the definition is saved except for the Scheduling scheme chosen which should be determined before every test.



λ * : 0.5188794598791836 ρ* is 1.9272298815467488 Not Stabilizable

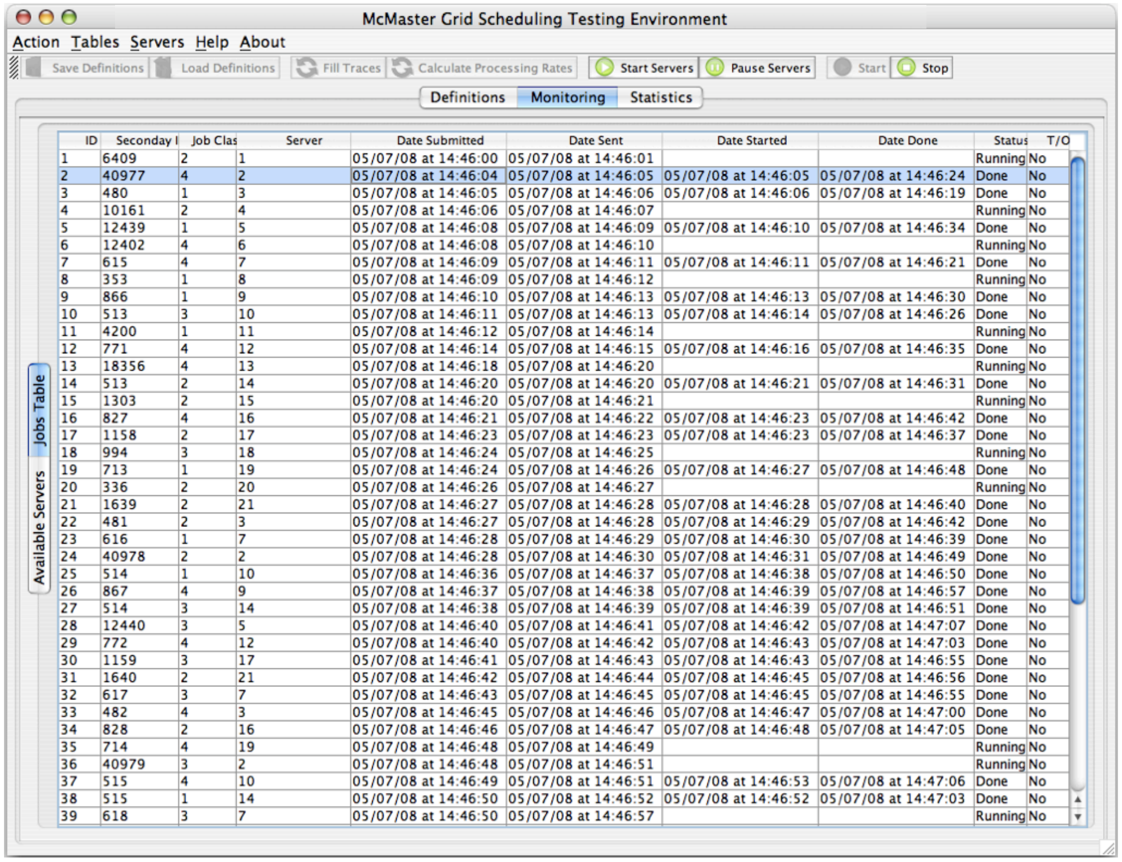
1	2	3	4	5	6
0.0	0.29770709010568186	0.0	0.0	0.0	0.0
1.0000000000000002	0.08541418021136288	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9388146183459196	1.0000000000000002
0.0	0.6168787296829553	1.0	1.0000000000000004	0.061185381654080384	0.0

Figure B.7: LP Sub Tab

B.4 Monitoring

After the completion of the definition phase, the experiment can be started by clicking on the *Start* button in the tool bar or *Action* in the menu bar. Under the *Monitoring* tab, there are two items to monitor: the Jobs Table and the Available Servers. In the

Jobs Table sub tab, jobs can be monitored. This table is updated whenever an event occurs. Under the *Available Servers* sub tab, the available servers can be monitored. To see the currently available servers, the *Update* button must be clicked to see the changes.



ID	Seconday I	Job Clas	Server	Date Submitted	Date Sent	Date Started	Date Done	Status	T/O
1	6409	2	1	05/07/08 at 14:46:00	05/07/08 at 14:46:01			Running	No
2	40977	4	2	05/07/08 at 14:46:04	05/07/08 at 14:46:05	05/07/08 at 14:46:05	05/07/08 at 14:46:24	Done	No
3	480	1	3	05/07/08 at 14:46:05	05/07/08 at 14:46:06	05/07/08 at 14:46:06	05/07/08 at 14:46:19	Done	No
4	10161	2	4	05/07/08 at 14:46:06	05/07/08 at 14:46:07			Running	No
5	12439	1	5	05/07/08 at 14:46:08	05/07/08 at 14:46:09	05/07/08 at 14:46:10	05/07/08 at 14:46:34	Done	No
6	12402	4	6	05/07/08 at 14:46:08	05/07/08 at 14:46:10			Running	No
7	615	4	7	05/07/08 at 14:46:09	05/07/08 at 14:46:11	05/07/08 at 14:46:11	05/07/08 at 14:46:21	Done	No
8	353	1	8	05/07/08 at 14:46:09	05/07/08 at 14:46:12			Running	No
9	866	1	9	05/07/08 at 14:46:10	05/07/08 at 14:46:13	05/07/08 at 14:46:13	05/07/08 at 14:46:30	Done	No
10	513	3	10	05/07/08 at 14:46:11	05/07/08 at 14:46:13	05/07/08 at 14:46:14	05/07/08 at 14:46:26	Done	No
11	4200	1	11	05/07/08 at 14:46:12	05/07/08 at 14:46:14			Running	No
12	771	4	12	05/07/08 at 14:46:14	05/07/08 at 14:46:15	05/07/08 at 14:46:16	05/07/08 at 14:46:35	Done	No
13	18356	4	13	05/07/08 at 14:46:18	05/07/08 at 14:46:20			Running	No
14	513	2	14	05/07/08 at 14:46:20	05/07/08 at 14:46:20	05/07/08 at 14:46:21	05/07/08 at 14:46:31	Done	No
15	1303	2	15	05/07/08 at 14:46:20	05/07/08 at 14:46:21			Running	No
16	827	4	16	05/07/08 at 14:46:21	05/07/08 at 14:46:22	05/07/08 at 14:46:23	05/07/08 at 14:46:42	Done	No
17	1158	2	17	05/07/08 at 14:46:23	05/07/08 at 14:46:23	05/07/08 at 14:46:23	05/07/08 at 14:46:37	Done	No
18	994	3	18	05/07/08 at 14:46:24	05/07/08 at 14:46:25			Running	No
19	713	1	19	05/07/08 at 14:46:24	05/07/08 at 14:46:26	05/07/08 at 14:46:27	05/07/08 at 14:46:48	Done	No
20	336	2	20	05/07/08 at 14:46:26	05/07/08 at 14:46:27			Running	No
21	1639	2	21	05/07/08 at 14:46:27	05/07/08 at 14:46:28	05/07/08 at 14:46:28	05/07/08 at 14:46:40	Done	No
22	481	2	3	05/07/08 at 14:46:27	05/07/08 at 14:46:28	05/07/08 at 14:46:29	05/07/08 at 14:46:42	Done	No
23	616	1	7	05/07/08 at 14:46:28	05/07/08 at 14:46:29	05/07/08 at 14:46:30	05/07/08 at 14:46:39	Done	No
24	40978	2	2	05/07/08 at 14:46:28	05/07/08 at 14:46:30	05/07/08 at 14:46:31	05/07/08 at 14:46:49	Done	No
25	514	1	10	05/07/08 at 14:46:36	05/07/08 at 14:46:37	05/07/08 at 14:46:38	05/07/08 at 14:46:50	Done	No
26	867	4	9	05/07/08 at 14:46:37	05/07/08 at 14:46:38	05/07/08 at 14:46:39	05/07/08 at 14:46:57	Done	No
27	514	3	14	05/07/08 at 14:46:38	05/07/08 at 14:46:39	05/07/08 at 14:46:39	05/07/08 at 14:46:51	Done	No
28	12440	3	5	05/07/08 at 14:46:40	05/07/08 at 14:46:41	05/07/08 at 14:46:42	05/07/08 at 14:47:07	Done	No
29	772	4	12	05/07/08 at 14:46:40	05/07/08 at 14:46:42	05/07/08 at 14:46:43	05/07/08 at 14:47:03	Done	No
30	1159	3	17	05/07/08 at 14:46:41	05/07/08 at 14:46:43	05/07/08 at 14:46:43	05/07/08 at 14:46:55	Done	No
31	1640	2	21	05/07/08 at 14:46:42	05/07/08 at 14:46:44	05/07/08 at 14:46:45	05/07/08 at 14:46:56	Done	No
32	617	3	7	05/07/08 at 14:46:43	05/07/08 at 14:46:45	05/07/08 at 14:46:45	05/07/08 at 14:46:55	Done	No
33	482	4	3	05/07/08 at 14:46:45	05/07/08 at 14:46:46	05/07/08 at 14:46:47	05/07/08 at 14:47:00	Done	No
34	828	2	16	05/07/08 at 14:46:46	05/07/08 at 14:46:47	05/07/08 at 14:46:48	05/07/08 at 14:47:05	Done	No
35	714	4	19	05/07/08 at 14:46:48	05/07/08 at 14:46:49			Running	No
36	40979	3	2	05/07/08 at 14:46:48	05/07/08 at 14:46:51			Running	No
37	515	4	10	05/07/08 at 14:46:49	05/07/08 at 14:46:51	05/07/08 at 14:46:53	05/07/08 at 14:47:06	Done	No
38	515	1	14	05/07/08 at 14:46:50	05/07/08 at 14:46:52	05/07/08 at 14:46:52	05/07/08 at 14:47:03	Done	No
39	618	3	7	05/07/08 at 14:46:50	05/07/08 at 14:46:57			Running	No

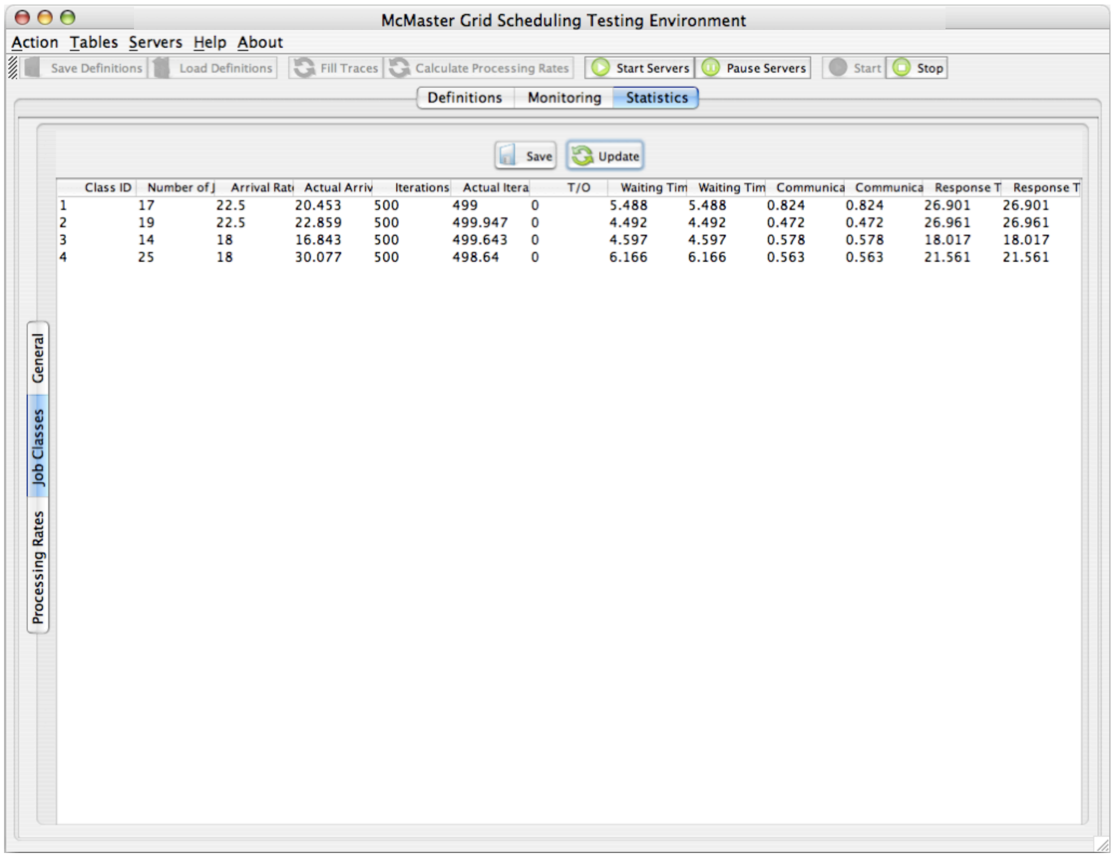
Figure B.8: Jobs Table

B.5 Statistics

To obtain statistics about the tests, the main tab *Statistics* is used. This main tab has three sub tabs:

- *General* sub tab which shows general statistics such as: the start time of the test, the time units elapsed and the response time.
- *Job Classes* sub tab which shows statistics about each job class. Such statistics include the average response time, average waiting time, total number of jobs arrived, desired arrival rate and actual arrival rate (Figure B.9).
- *Processing Rates* sub tab which shows the μ matrix and the actual processing rates per machine per job class.

All these statistics can be saved into files. This can be done using the *Tables* menu in the menu bar. A save dialogue appears on the screen. The user can browse to the target folder and then type the name of the test (e.g. LPAS). As a result four files will be saved (e.g. LPAS_classesStats.txt, LPAS_jobs.txt, LPAS_mue.txt, LPAS_systemStats.txt). The four files can be open with spread sheet applications. iWork 08 Numbers is recommended to process these files.



Class ID	Number of J	Arrival Rat	Actual Arriv	Iterations	Actual Itera	T/O	Waiting Tim	Waiting Tim	Communica	Communica	Response T	Response T
1	17	22.5	20.453	500	499	0	5.488	5.488	0.824	0.824	26.901	26.901
2	19	22.5	22.859	500	499.947	0	4.492	4.492	0.472	0.472	26.961	26.961
3	14	18	16.843	500	499.643	0	4.597	4.597	0.578	0.578	18.017	18.017
4	25	18	30.077	500	498.64	0	6.166	6.166	0.563	0.563	21.561	21.561

Figure B.9: Job Classes Statistics

Appendix C

Instructions

This appendix is a collection of setting configuration procedures to help in using the testing environment.

C.1 Creating an Xgrid controller/agent machine

To set a Mac OS X based machine as a server in a testing grid the following steps must be taken:

- **Setting a Password**. To do so one should:

  1. Open *System Preferences* under /Applications/System Preferences.app.
  2. Click on *Sharing* under *Internet & Network*. See Figure C.1.
  3. From the list of services click on *Xgrid* and click the Configure Button. See Figure C.2.
  4. Change the *Authentication Method* to *Password* and insert the password desired, then click on the *OK* button. This step will result in creating a file named */etc/xgrid/agent/controller-password* containing the password.
  5. Open the terminal and type *sudo cp /etc/xgrid/agent/controller-password /etc/xgrid/controller/agent-password*.

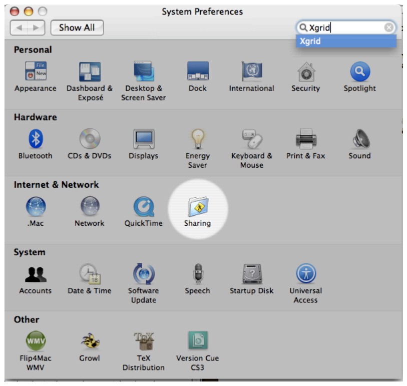
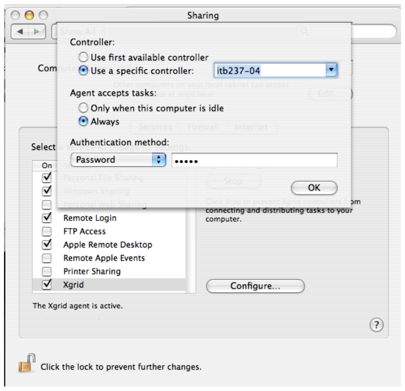


Figure C.1: The *Sharing* Tab in *System Preferences*

6. On the command-line, type *sudo cp /etc/xgrid/agent/controller-password /etc/xgrid/controller/client-password.*

- **Starting the agent and the controller**.

1. Execute the following: *sudo xgridctl c start.* This results in starting the controller process.
2. Open *System Preferences* under /Applications/System Preferences.app.
3. Click on *Sharing* under *Internet & Network*. See Figure C.1.
4. From the list of services click on *Xgrid* and click the Configure Button. See Figure C.2.
5. Click on the *Use a specific controller* radio button and choose or type the machine's full host name.
6. Click the *OK* button.
7. Check the *Xgrid* check button from the Services list.


Sharing

Controller:
( ) Use first available controller
(•) Use a specific controller: itb237-04

Agent accepts tasks:
( ) Only when this computer is idle
(•) Always

Authentication method:
Password •••••

On	
✓	Remote Login
	FTP Access
✓	Apple Remote Desktop
	Remote Apple Events
	Printer Sharing
✓	Xgrid

connecting and distributing tasks to your computer.

The Xgrid agent is active.

Click the lock to prevent further changes.

Figure C.2: The list of services and the *Configure* button.

C.2 Setting the Xgrid agent to execute only one task at a time

To configure the Xgrid agent to execute only one task at a time the following should be done:

- Open the following file: */Library/Preferences/com.apple.xgrid.agent.plist*
- Edit the MaximumTaskCount and set it to 1.
- Restart the agent process for this change to take effect.

Appendix D

LoopUsed As a process

```
static public void main(String [] args) {
        /*
                This main methods takes two arguments.
                        The first one is the iterations for the triple loop
                        The second on is a ratio used to control the length of the process
                            .
        */
        System.out.println("Statring");
        int iterations;
        double ratio = Double.parseDouble(args[1]);
        for (int k = 0; k < 100*Double.parseDouble(args[1]); k++){ // 100* ratio
             iterations
                iterations = Integer.parseInt(args[0]);
                int dum = 0; // dummy variable
                for (int i = 0; i < iterations; i++){  // The triple loop.
                        for (int j = 0; j < iterations; j++) {
                                for (int m = 0; m < iterations; m++); {
                                        dum++; // meaningless operation
                                        dum--;  //  another meaningless operation
                                }
                        }
                }
                /* The following output is used by the Xgrid layer to determine the
                     percentage don from the job. */
                System.out.println("<xgrid>{control␣=␣statusUpdate;␣percentDone␣=" + (k+1)
                     *ratio + ";␣}</xgrid>");
        }
        System.out.println("Done");
    }
```

Appendix E

Statistics

E.1 Ratios Statistics

The source code of the jobs sent to servers can be found in Appendix D. Every job is constructed from three nested loops that are executed $100x$ times, where x is called the ratio. If x equals 1 then the three nested loop will be executed 100 times, and if x equals 0.5 the three nested loops will be executed only 50 times. To impose heterogeneity this ratio is changed. For example, if machine *itb237-01.cas.mcmaster.ca* can execute a loop process with 500 iterations at rate 2 tasks per time unit, and the tester sets the assumed rate as 4 tasks per time unit then the ratio is 0.5. The following figures show the relationship between the ratios and the execution time for a job with 500 iterations.

E.2 Availability Statistics

The relation between $\mu'_{i,j} = a_j \mu_{i,j}$ [3] was found to hold more accurately on Intel Mac-based machines than on Power PC Mac-based machines. However on both types of hardware the relationship holds when $a_j > 0.5$. As a result, i2n our experiments we used values of a_j that are larger than 0.5. The following figures show the relation between a_j and the execution rates. The Rate units are tasks per second.

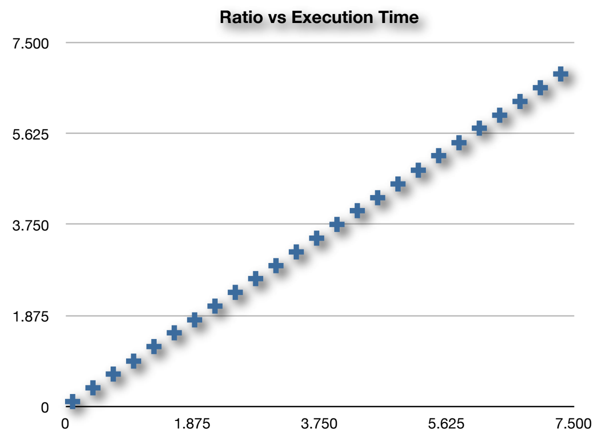


Figure E.1: Power-PC Machine (itb237-01)

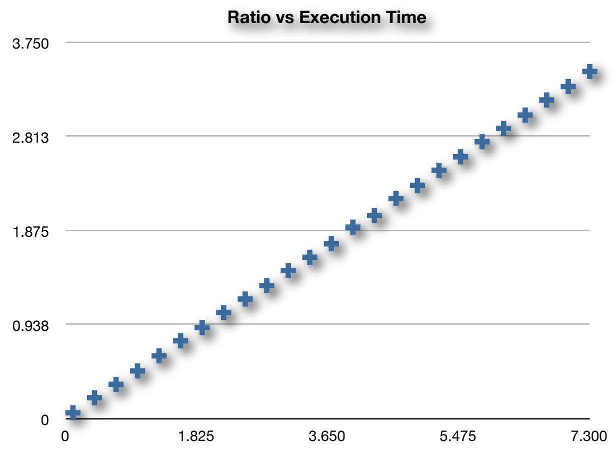


Figure E.2: Intel-based Machine (itb237-04

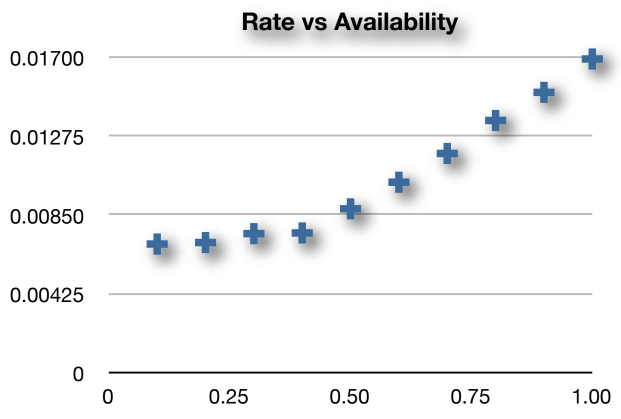


Figure E.3: Power-PC Machine (itb237-01)

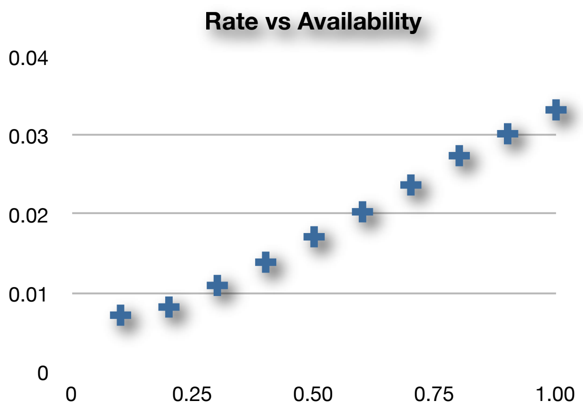


Figure E.4: Intel-based Machine (itb237-04)