INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

A Decentralized Utility-based Grid Scheduling Algorithm

João Luís Vazão Vasques

Dissertation submitted to obtain the Master Degree in
Communication Networks Engineering

Jury

President:	Prof. Fernando Henrique Corte-Real Mira da Silva
Supervisor:	Prof. Luís Manuel Antunes Veiga
Members:	Prof. João Carlos Serrenho Dias Pereira

November 2012

Acknowledgments

First of all, I would to thank to my supervisor, Professor Luís Veiga for all the support and guidance during my master thesis. His advices and fruitful discussions were very enlightening and made me learn a lot.

A special words to some professors that were very important for me as a student. Professor Miguel Abreu, Guilherme Arroz and Ana Moura Santos for the amazing courses and kindness. Professor Pedros Reis Santos for teaching how to succeed after failure. Professors Paulo Carreira and Ricardo Chaves for all the energy and support during one of the hardest period in my academic life. Professors Rui Rocha and Rui Valadas for their friendship and support in a very particular way. A special thanks to two other professors, Luis Rodrigues for all the encouragement, and Luis Caldas de Oliveira for his wise advice.

To my friends for all their friendship, fun and hard work. My "godfather" António Fonseca for being there in the most stressful moments. To Rui Costa, João Andrade, Mário Nzualo, Fábio Domingos and Bernardo Grilo, Sofia Vistas and Rita Raimundo for their friendship and some very good moments. To Afonso Oliveira, Carlos Simões and Henrique Fonseca for their loyalty and unforgettable adventures. To Bernardo Simões for fourteen years of friendship. To Xavier Azcue, Patricia Ribeiro, Mafalda Laranjo and Catarina Vieira for some of the great moments outside of Técnico.

To Cristina Fonseca and Raoul Felix for their inspiring company and support during this last year.

Last but not least, to my family. To my parents for all the love, support and comprehension. To my brother for always being by my side. To my grandparents for all the patience, love and courage.

This work was partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/108963/2008 and PEst-OE/EEI/LA0021/2011.

November 2012

João Luís Vazão Vasques

To my parents, brother and grandparents

Resumo

Os sistema grid têm ganho uma enorme importância nos últimos anos desde que os requisitos das aplicações aumentaram drasticamente. A heterogeneidade e dispersão geográfica dos recursos coloca alguns desafios difíceis, como o escalonamento de tarefas. Um algoritmo de escalonamento tenta encontrar um recurso para uma tarefa que preencha os requisitos da mesma, optimizando uma função objectivo. A utilidade é uma medida de satisfação do utilizador, que pode ser vista como uma função objectivo que um escalonador tenta maximizar.

Muitas funções de utilidade foram propostas para algoritmos de escalonamento. No entanto, os algoritmos propostos não consideram a satisfação parcial de requisitos , atribuindo um valor de utilidade baseado na total satisfação do mesmo. A maioria das soluções propostas segue uma abordagem centralizada ou hierárquica. Estas soluções sofrem de problemas de escalabilidade e tolerância a faltas.

Esta tese propõe uma arquitectura de escalonamento descentralizada com algoritmo de escalonamento com utilidades, que considera a satisfação parcial dos requisitos de modo a superar as limitações de soluções reais.

Abstract

Grid systems have gain tremendous importance in past years since application requirements increased drastically. The heterogeneity and geographic dispersion of grid resources and applications places some difficult challenges such as job scheduling. A scheduling algorithm tries to find a resource for a job that fulfills the job's requirements while optimizing a given objective function. Utility is a measure of a user's satisfaction that can be seen as an objective function that a scheduler tries to maximize.

Many utility functions have been proposed as an objective for scheduling algorithms. However, the proposed algorithms do not consider partial requirement satisfaction by awarding an utility based on the total fulfillment of the requirement. Most of them follow a centralized or hierarchical approaches, suffering from scalability and fault tolerance problems.

This thesis proposes a decentralized scheduling architecture with utility based scheduling algorithm that considers partial requirements satisfaction to overcome the shortcomings of actual solutions.

Palavras Chave
Keywords

Palavras Chave

Grelha

Escalonamento

Utilidade

Arquitectura descentralizada

Simulação

Keywords

Grid

Scheduling

Utility

Decentralized architecture

Simulation

Index

List of Figures

List of Tables

Abbreviations

BU Binary Utility

CPU Central Processing Unit

GIS Grid Information Service

ID IDentification

I/O Input/Ouput

JVM Java Virtual Machine

MM Matchmaking

MI Million Instructions

MIPS Millions Instruction Per Second

MTU Maximum Transmission Unit

QoS Quality of Service

PE Processor Elements

PU Partial Utility

RAM Random Access Memory

RR Round Robin

TTL Time-To-Live

UAP Utility Allocation Policy

VO Virtual Organization

1 Introduction

1.1 Historical overview

The idea of having access to computational power as we have to electricity is not new. In 1961, John McCarthy stated that "computation may someday be organized as a public entity." The term "Grid" was chosen because of the parallel that was made to the electric grid. An electrical grid provides resources, e.g. electricity, to many heterogeneous entities in a distributed and geographic dispersed environment. Grid computing follows the same principle but with different participants since it provides computational power to users instead of electricity. A Grid is formed by many heterogeneous resources. Sets of resources that share common sharing rules and conditions are called Virtual Organizations (VO).

Grid computing had its breakthrough on the 1990's which coincided with the boom of the Internet. The massification of the internet combined with the constant increase of network bandwidth computing power of devices (see the transistors Moore's law) and decrease of resources cost opened the doors to Grid computing.

The computational power required by science nowadays is huge. Genetic studies, large macroeconomic simulations and physicists trying to find the origin of the universe are examples of investigation areas that need access to a lot of computational power, e.g. computational resources. Due to the continuous growing need of science for computational resources it is important to have mechanisms that assure that shared resources are used in an efficient and fair way. For this reason, grid scheduling is a very important problem that has been widely studied by the computer science community. The purpose of grid scheduling is to allocate a job to a resource, fulfilling the job's requirements while optimizing resource utilization.

1.2 Problem statement

Effective scheduling is crucial in large scale systems such as grids and so, many scheduling solutions have been purposed. However, most of them use a centralized or hierarchical approach, lacking for scalability and reliability. Some decentralized approaches have also been proposed, but they do not consider partial requirement fulfillment, as scheduling policies are very strict and job requirements lack a proper characterization. Some of these solutions do not consider Quality of Service (QoS) or utility.

Those that consider utility and QoS do not have very flexible solutions concerning job's requirements fulfillment, which have a major impact on the network performance and users' satisfaction.

Considering this context, the main question that we want to address in this thesis is:

- Is it possible to devise a decentralized scheduling architecture where the scheduling decisions take into account the grid resources and the user's requirements in order to improve the network performance and the users' satisfaction?

1.3 Thesis objectives and expected contributions

The goal of this thesis is to design a new decentralized utility based scheduling algorithm for grid environments that will incorporate partial requirement fulfillment based on user's requirements. With this new scheduling approach, we pretend the optimize resource utilization and maximize user's utility.

The main contributions of this thesis are:

- A new decentralized scheduling architecture that provides partial utility fulfillment that, for the best of our knowledge, no other utility based grid scheduling algorithm uses.
- An extensive set of extensions to GridSim simulator for the creation of distributed architectures.
- A new resource allocation policy in GridSim for the support of priority multi-task environment.
- A set of modifications to GridSim for the support of our different scheduling algorithms.
- A validation of the proposal by simulation in a variety of conditions.

A set of results of this thesis have been submitted to publication, at the 28th Symposium On Applied Computing - SAC-2013. The paper is: J. Vasques and Luis Veiga. "A Decentralized Utility-based Grid Scheduling Algorithm", 28th Symposium On Applied Computing - SAC-2013" (Vasques & Veiga 2013).

1.4 Decentralized scheduling architecture overview

The decentralized architecture organizes the grid network into VOs, each one of them comprising a set of resources and a local scheduler. Users submit their jobs at their local scheduler and define the jobs' requirements, ranking the available options in terms of partial utility. Jobs may be submitted locally or remotely, depending on the available resource that bests fits their needs. To do so, schedulers maintains a snapshot of the grid resource status, by periodically or event-driven, exchange resource status information with other VOs. Local information is kept up-to-date, as it is updated whenever a

job is submitted or completed. Whenever a remote submission is performed, the job is forwarded to the remote scheduler, which has the final decision of accepting it or not, depending on the availability of the required resource.

1.5 Organization of the thesis

This thesis comprises six chapters and one appendix. At the beginning of each chapter there is a small introduction that describes its organization. At the end of each chapter, there is a section that synthesizes the relevant topics that were presented. The appendix was created due to number of graphics that were obtained from the simulations. It is also important to mention that all the entities are written in italic.

Chapter 1 introduces the problem that is addressed in the thesis in an historical perspective, presents the thesis objectives and expected contributions, and a brief overview of the proposed solution. At the end of this chapter, in this section, a brief overview of the thesis organization is provided to guide the readers.

Chapter 2 presents the thesis' related work. Section 2.1 presents and describes relevant grid middleware systems. At the end of this section a comparative analysis and classification of each one of the grid middleware systems is presented. Section 2.2 describes the most important grid simulators. At the end of this section a classification of each simulator is presented. Section 2.3 presents the most important aspects of resource discovery and some implementations. A classification of each type of resource discovery approach is done in the end of the section. Section 2.4 describes various aspects grid job scheduling such as the scheduling's phases, classes of scheduling algorithms and examples of grid scheduling algorithms. At the end of this section a classification of the described algorithms is done.

Chapter 3 presents the proposed solution. Section 3.1 presents the system architecture and some important topics such as requirement analysis, design options, the organization of the grid and the VOs. Section 3.2 presents and details the modules that were designed. These important modules are the grid resources, the scheduler and the its important submodules, the Resource Manager and Job Scheduler. Section 3.3 describes how users can specify job's requirements , our utility function and how the utility value is calculated. Section 3.4 presents a set of algorithms that were used for comparison purposes. Section 3.5 synthesizes the content of the chapter.

Chapter 4 presents the implementation issues. Section 4.1 presents and describes the architecture and relevant entities of the simulator. Section 4.2 presents and describes the extensions that were added to the simulator. The extensions include completely new modules that were developed, described in section 4.2.1, and some additions that were made to existing modules. Section 4.3 presents how topologies are created in the simulator. Finally, Section 4.4 synthesizes the content of the chapter.

Chapter 5 presents the simulation studies. Section 5.1 describes the simulation environment, the main goals and the setup. Section 5.2 presents the simulations aimed to compare the performance of the different scheduling algorithm: the methodology, metrics and results of the different scenarios are described. Aftter that, a preliminary validation of our decentralized architecture is performed in section 5.3. The final section 5.4 summarizes the results achieved and provides some final remarks.

Chapter 6 presents the conclusions of this thesis and some future work.

2 Related Work

This chapter describes the most relevant research work for the definition of the utility-based scheduler, organized according to a top-down approach. Section 2.1 describes some grid middleware solutions. Section 2.2 presents and describes some grid simulators. Section 2.3 describes how resource discovery is performed is grids. Section 2.4 describes important scheduler aspects, such as scheduling phases, classes of algorithms and some scheduling algorithms., metrics and criteria for grid scheduling. Finally, section 2.5 some of the issues that lead to the proposal of the algorithm are discussed.

2.1 Grid Middleware

Middleware aims to provide abstractions to programmers by shielding them from the complexity of the grid. Next we present some of the most important grid middleware systems. We will cover some important aspects of grid middleware such as how an application is defined, what kind of applications are supported, what is the grid architecture that the middleware was built for.

2.1.1 Condor-G

Condor-G (Frey et al. 2002) aims at providing users access to computational resources at many sites, which is a challenging issue due to the wide variety of grid resources.

Condor-G combines the intra-domain resource and computational management methods of Condor (Thain et al. 2003) with the inter-domain resource management protocols of the Globus Toolkit [1] . From Condor comes important aspects related to intra-domain resource and job management, such as resource discovery, job submission, job allocation and scheduling (Imamagic et al. 2006b). From Globus project it uses the Globus Toolkit protocols to address the remote resource access issue, namely: the Grid Security Infrastructure (GSI) (Foster et al. 1998) for authentication and authorization allowing the system to authenticate a user just once; the Grid Resource Allocation and Management (GRAM) (Czajkowski et al. 1998) for remote submission of a computational request; the Monitoring and Discovery System (MDS) (Czajkowski et al. 2001) for getting information about grid resources; and the Global Access to Secondary Storage (GASS) (Bester et al. 1999) for data transfer.

[1] www.globus.org

The Condor-G agent, also named computational management service, allows the user to treat the grid as a local resource and gives the possibility to perform operations such as submitting jobs, query a job's status, be informed of job termination and access job's logs. Condor-G agent can be accessed by a personal desktop agent, which uses the Globus protocols described above to interact with the machines on the Grid.

Jobs and resources are announced through the use of ClassAds, a set of uniquely named expressions, e.g. attributes formed by pairs of (name, values), that is assembled using a semi-structured data model (Thain et al. 2003). Condor-G supports two kinds of application types: Bag-of-Tasks (BOT) and Message Passing Interface (MPI). A BOT application consists of multiple independent tasks with no communication among each other. A MPI application is composed of multiple tasks with inter-task communication. When a user submits a job, the job is passed to the agent's scheduler, which is responsible for job scheduling, monitoring, fault-tolerance and credential management (Rahman et al. 2011). The scheduling operations are performed using the Matchmaking mechanism (Frey et al. 2002; Imamagic et al. 2006a) using a centralized matchmaker. Resources and users express their characteristics and the constrains to the matchmaker. The matchmaker uses the information from the ClassAds to select the appropriate resource. The scheduler creates a Condor-G GridManager deamon which is responsible for managing and submitting all the jobs of a single user. The GridManager terminates when all the user's jobs are completed. Each job submission request of the GridManager results in the creation of a Globus Job Manager on the selected resource(s). Condor-G follows a centralized scheduling approach.

In (Jacob et al. 2011), Jacob et al. propose a multi dimensional matchmaking framework to overcome some of the Condor-G's matchmaking shortcomings, such as lack support for parallel jobs and non-consideration of dynamic information.

2.1.2 Nimrod-G

Nimrod-G (Buyya et al. 2000) is a Grid middleware for building and mananing large computacional experiments over distributed resources (Rahman et al. 2011) that supports dealine and economy-based computations. Experiments are described using a simple declarative parametric modeling language (DPML). Nimrod-G supports BOT and MPI applications.

Nimrod-G uses the Globus (Foster & Kesselman 1996) middleware services for dynamic resource discovery and dispatching jobs. The main components on the Nimrod-G architecture are: Client or User Station, Parametric Engine, Scheduler, Dispatcher and Job-Wrapper.

The User Station is a user-interface whose function is to control and supervise a specific experience (list status of all jobs). The user can vary time and cost parameters while the scheduling is taking place.

The Parametric Engine receives the experiment plan described by the declarative parametric mod-

eling language and is responsible for maintaining the state of the experiment, creation of jobs, maintenance of job status and interacting with clients, scheduler and dispatcher.

The Scheduler is responsible for resource discovery, resource selection and job assignment. Nimrod-G's Scheduler is organized in a hierarchical way unlike Condor-G which follows a centralized approach. The resource discovery algorithm interacts with the Grid Information Service to get a list of the authorized machines and to keep track of the resources status. Nimrod-G scheduling approach is based on computational economy (Rahman et al. 2011; Abramson et al. 2002; Buyya et al.). Nimrod-G was one of the first grid middleware using the computational economy approach to grid scheduling. Computational economy can be handled in two ways. First, the system can work on the user's behalf and try to complete the assigned work within a given timeline and cost. If the system cannot satisfy the user's request then the user is informed. Second, the user can negotiate for resources and find out if the job can be performed. Important parameters of computational economy are the resource cost (set by its owner), the price the user is willing to pay and the deadline for the execution completion. Nimrod-G's scheduling objective is to maximize utility.

The Dispatcher initiates the execution of a task on the resource selected by the Scheduler and periodically updates the task execution status to the Parametric Engine.The Dispatcher is also responsible for starting the Job-Wrapper.

The Job-Wrapper responsible for setting up the environment on the selected resource for a task (Abramson et al. 2002), starting the execution of the task on the selected resource and sending the results back to the Parametric Engine via Dispatcher.

2.1.3 Askalon

Askalon (Fahringer et al. 2005) is a Grid middleware for application development and computing environment whose goal is to provide an invisible Grid to application developers. Askalon provides four tools to the user: Scalea, Zenturio, Aksum and PerformanceProphet.

Scalea (Truong & Fahringer 2002) is a performance instrumentation, measurement, and analysis tool. Zenturio (Prodan & Fahringer 2004) is a tool designed to specify and automatically conduct large sets of experiments, supporting multi-experience performance analysis. Aksum (Fahringer & Seragiotto 2002) is a tool for performance analysis that helps programmers to understand performance problems such as message passing and mixed parallel programs. The PerformanceProphet helps the users in terms of modeling and predicting the performance of behavior of distributed and parallel applications. Unlike other middleware systems such as Condor-G and Nimrod-G, Askalon is designed as a set of distributed grid services using web services.

Askalon supports workflow applications. A workflow application can be modeled as a Direct

Acyclic Graph (DAG) where the tasks are the nodes and the dependencies between tasks are the arcs among the nodes. The user can describe workflows using the XML-based Abstract Grid Workflow Language (AGWS) (Fahringer et al. 2005). Askalon's Resource Manager, GridARM (Fahringer et al. 2007), provides user authorization, resource management, resource discovery and advanced reservation. Resource discovery and matching are performed based on the constrains provided by the Scheduler.

Askalon's Scheduler has a centralized architecture and processes the workflow specification described in AGWL, converts it to an executable form and maps it onto available resources (Fahringer et al. 2007). The Scheduler uses GridARM to get information about the Grid resources and maps the workflow onto resources using a Genetic Algorithm based on user-defined QoS parameters. After that, a dynamic scheduling algorithm takes into consideration aspects such as machine crashes or CPU and network load and performs a reschedule if necessary. The Execution Engine is responsible for controlling the execution of a workflow based on information provided by the Scheduler.

2.1.4 Pegasus

Pegasus (Deelman et al. 2005; Deelman et al. 2004) is part of the GridPhyN project (Zhao et al. 2006) and is a system that maps complex scientific workflows onto Grid resources.

Pegasus uses the Globus Toolkit (**?**) GRAM (Czajkowski et al. 1998) for remote job submission and management; Monitoring and Discovery Service (MDS) (Czajkowski et al. 2001) to get information about the state of resources; Replica Location Service (RLS) (Chervenak et al. 2002) to get information about the data available at the resource.

Pegasus uses DAGMan and Condor-G (Frey et al. 2002) to submit jobs on Globus-based resources. There are two main components in Pegasus: Pegasus Workflow Mapping Engine (PWME) and DAGMan workflow executor for Condor-G. PWME receives an abstract workflow description and generates an optimized concrete workflow . An abstract workflow describes the computation in terms of logical files and logical transformations and indicate the dependencies between the workflow components and can be described using Chimera's (Foster et al. 2002) Virtual Data Language (VDL). A concrete workflow is an executable workflow that DAGMan can process. First, Pegasus queries the MDS to get information about the availability of the resources. The next step consists in reducing the workflow to only contain the necessary tasks for the final product. This is done by querying the RLS for replicas of the required data. Next, Pegasus queries the Transformation Catalog (TC) to find the location of the logical transformation (software components) defined on the workflow. The information obtained is used to make scheduling decisions (random selection, round robin, min-min). It is possible add new scheduling algorithms to Pegasus. Pegasus has an option that clusters jobs together in case they are small jobs assigned to the same resource. The information about the application and the selected resources is used to build

a concrete workflow which is sent to DAGMan. DAGMan will follow the dependencies of tasks and submit to Condor-G that will dispatch them to selected resources.

2.1.5 Comparison

Table 2.1 summarizes the most relevant properties of the different middleware systems that were described. Using information from (Rahman et al. 2011) five categories have been chosen to make the classification: Application Type, Application Definition, Scheduling Architecture, Scheduling Objective and Self-Optimization. The Scheduling Objective has two categories identified by the letters *d* and *r* meaning deadline and resource utilization respectively.

The Application Type category states for the dependency between tasks. Applications can be divided in three types: Bag-of-Task (BOT), Message Passing Interface (MPI) and Workflow. A BOT application consists of multiple independent tasks with no communication among each other. A MPI application is composed of multiple tasks with inter-task communication. A workflow application can be modeled as a Direct Acyclic Graph (DAG) where the tasks are the nodes and the dependencies between tasks are the arcs among the nodes.

The Application Definition category refers to the different definition languages and tools that users can use to define applications. Each system has its own approach, Condor-G uses ClassAds, Nimrod-G uses a Declarative Parametric Modeling Language (DPML), Askalon uses Abstract Grid Workflow Language (AGWL) and Pegasus uses Chimera's Virtual Data Language (VDL).

The Scheduling Architecture category is related, as the name implies, with the architecture of the scheduling infrastructure and can be divided into three types: Centralized, Hierarchical and Decentralized. In a centralized architecture, scheduling decisions are made by a central controller that maintains information about all applications and resources in the system. In a hierarchical architecture, there is a central manager and multiple low-level schedulers. The manager is responsible for handling task execution and assigns individual tasks to low-level scheduler that are responsible for mapping them onto resources. In a decentralized architecture the number of tasks managed by one scheduler is limited. This solution is more fault-tolerant and scalable than the other two but has some challenges regarding security and information management.

The Scheduling Objective category presents some of the goals of the scheduling, namely: utility, optimization and load balancing. Utility is a measure of relative satisfaction. Optimization is related to improvement of performance regarding completion time or resource utilization. Load balancing is also a measure of performance, related to the distribution of workload between resources to avoid resource overload.

Systems	App. Type	App. Definition	Scheduling Architecture	Scheduling Objective
Condor-G	BOT/MPI	ClassAd	Centralized	Load Balancing
Nimrod-G	BOT/MPI	DPML	Hierarchical	Utility/Optimization(d)
Askalon	Workflow	AGWL	Centralized	Utility/Optimization(r)
Pegasus	Workflow	VDL	Centralized	Optimization(r)

Table 2.1: Middleware classification

2.2 Grid Simulators

Evaluation and comparative analysis of grid scheduling algorithms and research experiments are often difficult to perform. This is caused by many problems, including, for example, difficulties in obtaining exclusive access to large scale infrastructures for research purposes or lack of certain functionalities of real resource management systems, such as advance reservation (AR) or Grid user accounting. Therefore, Grid scheduling algorithms have been often tested in simulation environments. Simulators are useful to observe with high precision a local or global characteristic of a distributed system. The fundamental advantage of the simulators is their independence to the execution platform. Simulating a mechanism of a 10 000 nodes distributed system on a single PC is not rare. This advantage is made possible because the simulator does not run the real distributed system but a model of it. The rest of this section describes some of the most popular grid simulators.

2.2.1 Bricks

Bricks (Takefusa et al. 1999) was the first proposed Grid simulator designed for scheduling issues. Bricks was proposed and designed for studies and comparisons of scheduling algorithms and frameworks, under various structural and workload conditions. Bricks allows the simulation of various behaviors: resource scheduling algorithms, programming modules for scheduling, network topology of clients and servers in global computing systems, and processing schemes for networks and servers. It is basically a Java discrete event driven simulator where users can specify network topologies, server architectures, communication models and scheduling framework components. Its is possible to add new scheduling features by modifying a module called *Scheduling Unit*. Bricks has been used in experiences associated with the NWS (Network Weather Service) and for High-Energy Physics. (Takefusa et al. 1999). According to *Google Scholar*, the Bricks paper (Takefusa et al. 1999) has been cited 114 times.

2.2.2 SimGrid

SimGrid (Legrand et al. 2003) was developed to study single-client multi-servers scheduling in the context of complex, distributed, dynamic, heterogeneous environments. Since in its general form, the scheduling problem is NP complete. SimGrid is based on event driven simulation. Resources have

characteristics like speed, availability, etc., based on constant or traces and are modeled by their latency and service rate. It provides a set of abstractions and functionalities to build a simulator corresponding to the applications and infrastructures characteristics. These characteristics may be set as constants or evolve according to previously collected traces. The topology is fully configurable and jobs have a cost and a state. SimGrid is available in C, Java and recently, Ruby. The SimGrid paper (Legrand et al. 2003) has been cited 322 times according to *Google Scholar*.

2.2.3 GridSim

GridSim (Buyya & Murshed 2002) is a very popular Java Grid simulator used for scheduling purposes. Like SimGrid, GridSim is a discrete event simulator. It can be used to simulate application schedulers for single or multiple administrative domains distributed computing systems such as clusters and Grids. Resources are described with number of processors, cost of processing, performance, internal scheduling policy, workload, time zone. GridSim makes a difference between an entity's inputs and outputs, managing the two in a separate way. This provides a mean to express performance differences between parameters and results communication. GridSim is built on a modular to allow the implementation of new processes and behaviors to its entities as well as new scheduling policies. The simulator also provides tools for creating network topologies, statistical analysis and simulate resource failure. According to data from *Google Scholar*, the GridSim paper (Buyya & Murshed 2002) has been cited 1064 times.

2.2.4 GSSIM - Grid Scheduling Simulator

GSSIM (Kurowski et al. 2007) is a Java discrete event simulator based on GridSim (Buyya & Murshed 2002). GSSIM allows the creation of network topologies, resources and can simulate application schedulers just like GridSim does. However, GSSIM adds some extra features such as the possibility of creating workflows. A job can have one or more tasks. Those tasks can have dependencies among themselves. For example, task 4 cannot start executing while task 2 is not completed. The preceding constrains between tasks are called worjflow. Workload contains information about jobs, their structure, resource requirements, relationships, etc. GSSIM can process workloads that follow the Grid Workload Format. The GSSIM paper (Kurowski et al. 2007) had 28 according from *Google Scholar* information.

2.2.5 Resume

Table 2.2 presents a synthesis of all the simulators that were described.

Simulators	Programing Language	Network topologies	Workloads	Number of citations
Bricks	Java	Yes	No	114
SimGrid	C/Java/Ruby	Yes	No	322
GridSim	Java	Yes	No	1064
GSSIM	Java	Yes	Yes	28

Table 2.2: Classification of grid simulator

2.3 Resource Discovery in Grids

Resource discovery is the process of searching and locating resource candidates that are suitable for executing jobs. The dynamic and heterogeneous nature of the Grid makes efficient resource discovery a challenging issue. In this section we will present some of the most important approaches of resource discovery and some implementations.

2.3.1 Resource Description and Matching

- **Centralized** in the centralized category, resource discovery is performed by querying a unique server. Resource discovery solutions use a central server to discover resources. Condor-G's matchmaking mechanism (Frey et al. 2002; Imamagic et al. 2006a) uses a ”Matchmaker” (central server) where resources advertise their specifications and users their requirements using ClassAds. The ”Matchmaker” acts as a ”yellow page” that finds the appropriate set of resources for a user request. In (Kaur & Sengupta) Kaur et al. propose a centralized resource discovery mechanism for grids which relies on web services. The proposed solution has four main components: UDDI rich Query Model, Grid Web Services Description Language (GWSDL), SOAP and HTTP.The UDDI rich Query Model uses the UDDI standard to discover grid services by maintaining resource information as key-value pair in the UDDI database. GWSDL is an extended version of WSDL that is used to describe grid services. SOAP is used for communication between web services in the grid. HTTP provides an easy to use interface to post and get requests.

- **Hierarchical** resource discovery solutions use a hierarchy of servers to discover resources. Nimrod-G (Buyya et al. 2000) follows a hierarchical resource discovery approach. In (Gomes Ramos & Magalhaes Alves de Melo 2006) Ramos et al. propose a solution for resource discovery in grids based on Globus Toolkit (GT3) using web services. The authors propose a hierarchical topology that divides the grid into Virtual Organizations (VO). Each VO has master and slave nodes. The master nodes are responsible for updating the resource database and the slave nodes for retrieving information from resources under their master command. Resource discovery is performed using a configuration file which includes the requested resource information. The configuration is used to generate an XML file that is distributed to all the slaves and the resource

search is initiated. Each slave checks if the request is satisfied and returns the information to its master. This approach is more scalable and reduces the bottleneck problem when compared to the centralized one. However, single point of failure still exists since failure of one server may cause that a large part of the nodes become invisible to queries. Like in centralized solutions, web services are very used in hierarchical resource discovery.

- **Decentralized or P2P** resource discovery aims to overcome some limitations of centralized and hierarchical solutions such as scalability and fault tolerance. In (Chen et al. 2011) the authors propose a P2P three layer resource discovery model for grids. The model is built on a structured P2P system and uses a Distributed Hash Table (DHT) to map nodes and data objects to overlay network. When a client makes a request, it will be carried out to find the resource on the local grid. If the resource is found, the result will be returned to the user. If the resource is not found, the request will be send to another grid (node) using the P2P virtual layer between nodes. The first layer in the model is formed by the IS root node. The root node is just a node that uses the same resource requests as the other high performance nodes. The second layer is formed by super nodes. Each administrative domain needs to have its own super node, which is registered in the IS root node. The super node provides resource information about its domain, accepts tasks from the upper layer. The super node sends the its results to the root node. Authors propose Chord (Stoica et al. 2001) or Gnutella to manage the second layer. The third layer is composed of the various domains resources This layer can be managed using Chord or Gnutella. In (Ma et al. 2010) Ma et al. propose a resource discovery model with three layers. The bottom layer is the resource layer. Each resource of a virtual organization (VO), super peer, must register in its super peer. The intermediate layer is formed by super peers. Each super peer saves information from its peers (resources) and exchange that information with other super peers. The upper layer is formed by Super peer Agents. Super peer Agents are resource services of a region and can take charge of one or more similar super peers. The resource discovery uses a DHT and an Ant colony optimization (ACO) algorithm. When a specific resource can not be found in a VO, located via Chord (Stoica et al. 2001), managed by a super peer the solution uses ACO to find the peer which has the required resource.

2.3.2 Resource Discovery Algorithms Classification

Table 2.3 provides a synthesis and classification of resource discovery in grids. A higher number of ”+” means that an approach is more successful on a particular aspect than the others. The reliability aspect is measured in terms of single point of failure. The dynamism aspect is related to resources dynamically joining and leaving the grid.

	Centralized	Hierarchical	P2P
Scalability	+	++	+++
Dynamism	+	++	+++
Reliability	+	++	+++
Server Bottleneck	+++	++	+
Example	Condor-G	Ramos et al.	Chen et al.

Table 2.3: Resource Discovery classification

2.4 Grid Job Scheduling

2.4.1 Phases of Grid Job Scheduling

The Grid scheduling process can be divided into three main phases (Schopf 2004): resource discovery where a list of potential resources is created, system or resource selection were a set of resources is chosen and task execution where the tasks are executed and monitored. Figure 2.1 was taken from (Schopf 2004) and shows the three main phases and the steps that make them.

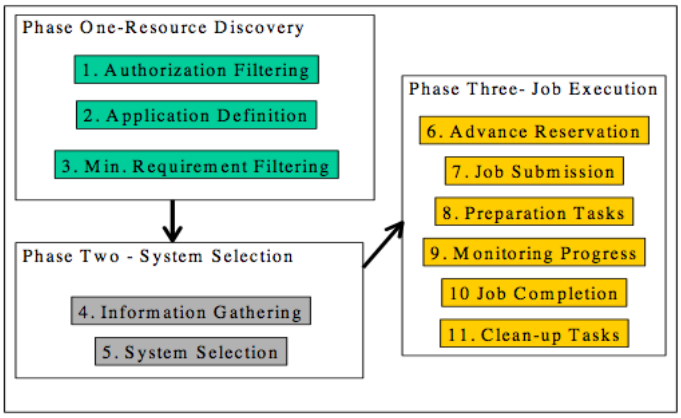

Figure 2.1: Grid scheduling phases

- **Resource Discovery:** the first stage is to know which resources are available. The first step consists in determining the set of resources that a user has access to. The is done consulting the Grid Information System (GIS). At the end of this step, the user will have a list of resources that he/she can access. The next step is the application requirement definition. In this step the user specifies a set of requirements for the job in order to filter the set of resources. An example of how requirements can be specified is the ClassAd used by Condor (Thain et al. 2003). The next step is to do a minimal requirement filtering. The goal is to eliminate the resources that do not meet the minimal requirements.

- **System Selection:** in this phase the goal is to select a single resource to schedule the job. This is

done in two steps: gather dynamic information and system selection. Gather dynamic information is important in order to make the best mapping between job and resource.

Information can be obtained by consulting the GIS and the local resource scheduler. The system selection consists in choosing a resource with the gathered information. One of the approaches to resource selection is Condor Matchmaking (Raman et al. 2000; Thain et al. 2003).

- **Task Execution:** the first step, advanced reservation, is optional. The goal is to make the best use of the system. Advanced reservation difficulty depends on the considered resource. When the resource or resources are chosen, the task needs to be submitted. Globus Grid Resource Allocation and Management (GRAM) is used by middleware systems such as Condor-G for job submission. The next step is preparation. In this step a set of operations take place to prepare the resource to run the task. The following step is monitoring. Once the task is started it is important to keep track of its progress. By monitoring tasks, the scheduler can conclude that a given task in not making progress and may reschedule it. The next step is job completion were the user is notified when a task or job finishes. The final step is the cleanup were temporary files are removed and the user collects information from the resource that will be used to analyze the results.

2.4.2 Classes of Scheduling Algorithms

There are many scheduling algorithms. In order to compare them and classify them, a classification needs to be made. In (Casavant & Kuhl 1988), Casavant et al propose a hierarchical taxonomy for scheduling algorithms in general-purpose parallel and distributed systems (Dong & Akl 2006). Grid falls into a subset of this taxonomy since it is a special kind of the systems that are considered in (Casavant & Kuhl 1988). Due to the nature of the Grid, some new characteristics such as batch, immediate, adaptive and preemptive scheduling need to be considered in Grid scheduling algorithms. In the following, there are described the main types of scheduling in Grids.

- **Local vs Global**: at the highest level, scheduling can be divided into local and global (Casavant & Kuhl 1988). Local scheduling operates on a single processor scenario. The scheduler is responsible for the allocation and execution of processes in the CPU (Dong & Akl 2006). Global scheduling allocate processes to multiple processors to optimize a system-wide performance goal (Dong & Akl 2006). Considering what was said before it is obvious to conclude that Grid scheduling is global.
- **Static vs Dynamic**: In (Xhafa & Abraham 2008) and (Xhafa & Abraham 2010), Xhafa et al. state that exist two main aspects to determine the dynamics of Grid scheduling: dynamics of job execution and dynamic of resources. Dynamics of job execution refers to the situation of job failure. Dynamics of resources refer to the possibility of resources join and leave the Grid and changes

of local resource usage policies. In static scheduling the information about the Grid's resources is available at schedule time, every task is assigned once to a resource and there are not job or tasks failures. With static scheduling it is possible to estimate computation costs before the task execution and to have a global view of costs and tasks (Dong & Akl 2006). These estimations cannot be done in scenarios with nodes can fail or become isolated. Since these situations can occur very often mechanisms such as rescheduling (Cooper et al. 2005) were introduced to smooth the problem.

In dynamic scheduling, cost estimation is difficult (Dong & Akl 2006), jobs can fail and resources can join and leave the Grid in an unpredictable way (Xhafa & Abraham 2008). Dynamic scheduling has two components: system state estimation and decision making. System state estimation is responsible for collecting information about the Grid and building an estimate. This estimate will be the base to the decision of mapping a task to a resource. Since it is not possible to estimate computation costs before execution load balancing is used as an alternative to ensure the system well functioning.

- **Centralized vs Decentralized vs Hierarchical**: The scheduling responsibility can be delegated on one centralized scheduler or be shared by multiple distributed schedulers. On the centralized approach there is only one scheduler for the Grid. In the centralized approach it is possible to monitor all the resources state which makes easier to create efficient schedulers (Xhafa & Abraham 2008). Another advantage of centralized scheduling is the easy management (Krauter et al. 2002) and implementation of schedulers. However, centralized scheduling approaches have a single point of failure (Xhafa & Abraham 2010), lack of scalability (Dong & Akl 2006; Krauter et al. 2002; Xhafa & Abraham 2008; Xhafa & Abraham 2010) and lack of fault-tolerance (Dong & Akl 2006; Krauter et al. 2002; Xhafa & Abraham 2008). Condor (Thain et al. 2003; Xhafa & Abraham 2010) uses a centralized scheduler based on the ClassAd matchmaker (Raman et al. 2000). On the decentralized approach there is no central scheduler that controls the resources. In this approach, local schedulers play an important role since the scheduling requests are sent to them. These type of schedulers take in consideration important issues such as fault-tolerance, scalability and multi-policy scheduling. In the hierarchical approach schedulers are organized in an hierarchical way. This approach is more scalable and fault-tolerant than the centralized approach although, it does not scale and it is not fault-tolerant as the decentralized approach.

- **Immediate vs batch**: In the immediate approach, jobs are schedule as they enter the system (Xhafa & Abraham 2010) using the system's scheduling algorithm. Jobs do not wait for the next time interval when the scheduler will get activated (Xhafa & Abraham 2010) On the other way, in the batch approach, jobs are grouped in batches and scheduled as a group (Xhafa & Abraham 2010). In the batch approach the scheduler can use job and resource characteristics better than immediate

schedulers since it has the time between the activation of the batch scheduler.

- **Adaptive**: This approach uses information regarding the current status of the resources and predictions of their future status to avoid a decrease of performance. Rescheduling is an adaptive scheduling where running jobs are migrated to other resources. In (Othman et al. 2003), Othman et al refer that the Grid must be able to recognize the state of resources and propose an adaptable resource broker. An example of an adaptive scheduling algorithm can be found on Huedo et al. work (Huedo et al. 2004).

Design Choice	Approaches
Dynamics	Dynamic Static
Architecture	Centralized Hierarchical Decentralized
Mode	Immediate Batch

Table 2.4: Classes of scheduling algorithms

2.4.3 Classic Scheduling Algorithms

In this section we present some of the classical scheduling algorithms in Grids and distributed systems. In the following algorithms m represents the number of resources and s the number of tasks in the meta-task.

First Come First Served

In First Come First Served algorithm, jobs are executed according to the arriving time order (Lee et al. 2011). This algorithm has a major disadvantage. When a large job is on the waiting queue, the jobs behind it must wait a long time for the large job to finish. This situation is called convoy effect.

Round Robin In the Round Robin algorithm each job is assigned a time interval, called quantum, during which it is allowed to run (Tanenbaum 2007). If a job cannot be completed in a quantum it will return to the queue and wait for the next round (Tanenbaum 2007). Round Robin has the advantage that a job does not need to wait for the previous job to complete to execute. The only challenging issue with this algorithm is to find a suitable length for the quantum (Tanenbaum 2007).

Minimum Execution Time The Minimum Execution Time (MET) algorithm assigns each task to the resource that performs it with the minimum execution time (Maheswaran et al. 1999). MET does not consider whether the resource is available or not at the time (ready time) (Etminani & Naghibzadeh 2007; Maheswaran et al. 1999; Sahu 2011) and can cause severe imbalance in load across resources (Etminani & Naghibzadeh 2007; Maheswaran et al. 1999; Sahu 2011). The main advantage of the algorithm is that

it gives to a task the resource that performs it in the smallest amount of time (Maheswaran et al. 1999). MET takes $O(m)$ time to map a task to a resource (Etminani & Naghibzadeh 2007).

Minimum Completion Time The Minimum Completion Time (MCT) algorithm assigns a task to the resource that obtains the earliest completion time for that task (Etminani & Naghibzadeh 2007; Maheswaran et al. 1999; Sahu 2011). MCT has the following disadvantage: the resource that was assigned to a task may not have the minimum execution time for it (Etminani & Naghibzadeh 2007; Maheswaran et al. 1999; Sahu 2011). MCT takes $O(m)$ time to map a task to a resource (Etminani & Naghibzadeh 2007).

Min-min The Min-min algorithm has two phases (Etminani & Naghibzadeh 2007). On the first phase, the completion time of all unassigned tasks on all available machines is used to calculate to minimum completion time of a task T on a machine M (Sahu 2011). On the second phase, the task with the minimum completion time is chosen, removed from the task list and assigned to the corresponding resource (Etminani & Naghibzadeh 2007). The process is repeated until all tasks are mapped to a resource. We can conclude that jobs that can be completed earlier have higher priority than the others (Izakian et al. 2009; Lee et al. 2011; Sahu 2011). Min-min takes $O(s^2m)$ time to map a task to a resource (Etminani & Naghibzadeh 2007).

Min-max The Min-Max algorithms has two phases (Izakian et al. 2009; Sahu 2011) and uses the minimum completion time (MCT) for the first phase and the minimum execution time (MET) for the second phase as metrics. The first phase of Min-Max is the same as the Min-min algorithm. The second phase the task whose $\frac{MET_{\{fastest\ machine\}}}{MET\{selected\ machine\}}$ has the maximum value will be selected for mapping (Izakian et al. 2009). The task is removed from the unassigned list, resource workload is updated and the process is repeated until the list is empty (Sahu 2011). The intuition of this algorithm is that we select resources and tasks from the first step that the resource can execute the task with a lower execution time in comparison with other resources (Izakian et al. 2009).

Max-min The Max-min has two phases as Min-min has(Etminani & Naghibzadeh 2007). The first phase is equal to the Min-min algorithm (Etminani & Naghibzadeh 2007; Izakian et al. 2009; Sahu 2011). On the second phase, the task with the maximum completion time is chosen, remove from the task list and assigned to the corresponding resource (Sahu 2011). The process is repeated until all tasks are mapped to a resource. Max-min can be combined with Min-min in scenarios where the are tasks of different lengths (Lee et al. 2011). Max-min takes $O(s^2m)$ time to map a task to a resource (Etminani & Naghibzadeh 2007).

Sufferage The sufferage of a task is the difference between its second minimum completion time and its first minimum completion time (Izakian et al. 2009; Sahu 2011). These completion times are calculated considering different resources (Lee et al. 2011). In the Sufferage algorithm the criteria to assign a task to a resource is the following: assign a resource to a task that would suffer the most if

that resource was not assigned to it (Lee et al. 2011; Maheswaran et al. 1999). The sufferage value of a task is the difference between its second earliest completion time and its earliest completion time (Lee et al. 2011; Maheswaran et al. 1999). Once a task is assigned to a resource it is removed from the list of unassigned tasks and the process is repeated until there are no tasks in the unassigned list. Sufferage takes $O(s^2m)$ time to map a task to a resource (Etminani & Naghibzadeh 2007).

2.4.4 Utility-based Scheduling Algorithms

QoS are constrains or bounds that are related to the provided service. QoS appeared on aspects related to telephony and computer networks such as service response time, loss, signal-to-noise ratio, cross-talk, echo, etc. In the Grid environment there are some different QoS aspects to consider such as deadline, price, execution time, overhead.

Utility is a concept, originally from economics, that evaluates the satisfaction of a consumer while using a service. In a Grid environment, utility can be combined with QoS constrains in order to have a quantitative evaluation of a user's satisfaction and system performance.

The classical scheduling algorithms presented in the previous section do not consider QoS or utility demands. Next, we present some QoS and utility scheduling algorithms for grid environments.

In (Amudha & Dhivyaprabha 2011) Amuda et al. propose a QoS priority-based scheduling algorithm. The algorithm assumes that all the necessary information about resources, jobs and priority values is available and is designed for batch mode independent tasks. The task partition divides the tasks into two groups (high and low) using the priority value as a QoS parameter. After the task division, the scheduler classifies the tasks into four categories: 1A - high complexity and high priority, 1B - low complexity and high priority, 2A - high complexity and low priority and 2B - low complexity and low priority. After task classification the resources are divided into two groups: high processing speed systems (group 1) and hybrid systems (group 2). High processing tasks go to group 1 and the others to group 2. Tasks with higher priority are scheduled first and equally on all the machines.

In (Chen 2010) Chen proposes an economic grid resource scheduling based on utility optimization that uses a universal flexible utility function that addresses QoS requirements of deadline and budget. The paper assumes that a grid is hierarchical and that the user submits the assignment to a Grid Resource Manager (GRM). The GRM is at the top of the hierarchy, on the second level there are the Domain Resource Managers (DRM) that are responsible for Computing Nodes (CN) or other DRM. The algorithm starts by the GRM getting utility information from all DRM and by calculating the rate of throughput and average response delay. Then, the algorithm finds out which of DRM has the maximum utility value (MUV) and selects it to be the scheduling node. If MUV is not unique, them the DRM which has the greatest variance is chosen. If the nodes on the next level of the chosen DRM are not CN them the

process is repeated. Otherwise, the algorithm finds the node which has the maximum utility value and gives it the user assignment.

In (Chunlin & Layuan 2007) Chunlin et al. propose an optimization approach for decentralized QoS-based scheduling based on utility and pricing. The authors consider two types of agents in the proposed scheduling model: Grid resource agents that represent the economic interests of the resources and Grid task agents that represent the interests of the Grid user. Grid resources can be divided into computational resources (CPU speed, memory size, storage capacity) and network resources (bandwidth, loss rate, delay and jitter). Task agents specify their resource requirements using a simple and declarative utility model. The Grid is seen as a market where the task agents act as consumers and resources as providers that compete with each other to maximize their profit. Due to the fact that the it is not realistic that the Grid knows all the utility functions of the task agents, although it is mathematical tractable, and it requires global coordination of all users, the authors propose a decomposition of the problem in two problems (task agent optimization and resource agent optimization) by adopting a computational economy framework. The proposed solution allows multi-dimensional QoS requirements that can be formulated as a utility function that is a weighted sum of each dimension's QoS utility function. Three QoS dimensions are considered: payment, deadline and reliability. The scheduling is done by solving the subproblems via an iterative algorithm. In each iteration, each player (task agent and resource agent) trade with each other to find a global optimum solution to the system while trying to maximize their own utility. The process stops when all arrive at the same solution.

2.4.5 Scheduling Algorithms classification

Table 2.5 presents a classification of all the scheduling algorithms described. The symbol * in the table means that there was not possible to evaluate the corresponding criteria of a particular algorithm using the information provided in the author's paper.

Table 2.5: Classification of scheduling algorithms

Algorithms	Order-Based	Heuristic	QoS	Utility	Mode	Complexity
FCFS	Yes	No	No	No	Batch	$O(1)$
Round Robin	Yes	No	No	No	Batch	$O(1)$
MET	No	Yes	No	No	Immediate	$O(m)$
MCT	No	Yes	No	No	Immediate	$O(m)$
Min-Min	No	Yes	No	No	Batch	$O(s^2m)$
Min-Max	No	Yes	No	No	Batch	$O(s^2m)$
Max-Min	No	Yes	No	No	Batch	$O(s^2m)$
Sufferage	No	Yes	No	No	Batch	$O(s^2m)$
Amuda et al.	No	Yes	Yes	No	Batch	*
Chen	No	Yes	Yes	Yes	*	*
Chunlin et al.	No	Yes	Yes	Yes	*	*

2.5 Synthesis and Final Remarks

In this chapter we presented different topics related to grid scheduling. First, we made a classification and description of the existing resource discovery approaches. Due to dynamics of the grid we concluded that the P2P approach was most adequate to grid environments. After resource discovery we presented grid scheduling. We started the section by presenting the stages of grid scheduling; then we presented some important criteria for classification of scheduling algorithms; next, we described and classified two different types of scheduling algorithms: no QoS or utility constrains and with QoS and/or utility constrains. Considering the algorithms with QoS and utility constrains we concluded that all solutions consider only complete requirement fulfillment when calculating utility.

In our solution we will consider scalability and bottleneck issues by proposing a P2P topology to the grid. Our utility scheduling algorithm will consider partial requirement fulfillment, i.e. partial utility.

3 Proposed Solution

This chapter describes the proposed solution. Section 3.1 overviews the system architecture, starting by the identification of requirements and design options, which is followed by the description of the grid organization and architecture. Section 3.2 describes the different modules that comprises the proposed solution. Section 3.3 describes how the user can specify job's requirements and the calculation of the utility function. A special attention is dedicated to the new utility function that was proposed and the entire section 3.3.2 is used to describe it. The other schedulers that are used to assess the performance of our algorithm are also described, in section 3.4. Finally, section 3.5 provides a synthesis of the chapter and discusses some relevant issues.

3.1 System Architecture

One of the most important aspects of our work is the selection of the type of grid architecture that will be used. This section describes it. We will start by analyzing the requirements and present the most relevant design options. Afterwards, we present the grid organization, introducing the concept of VOs and finally, we describe the architecture of a VO.

3.1.1 Requirement Analysis

The definition of the scheduler architecture is a key issue to provide users' satisfaction, as an inadequate design might lead to job failures or high execution times. Hence, several important requirements have been taken into account in the design of our solution.

- *Scalability* - the scheduler architecture must be able to support a very large number of resources and users creating concurrent jobs, without significantly compromising the grid network performance.
- *Reliability* - scheduler architecture must be defined in a way that, in case of scheduler failure, the grid remains available in order to keep accepting jobs for submission.
- *Multi-policy scheduling* - the schedule architecture must support different type of resources and different type of users' requirements without needing the realization of complex operations.

- *User satisfaction* - jobs need to be scheduled in a way that satisfies the majority of the users, meaning that their jobs are executed and their preferences are attended.
- *Performance* - jobs must also be completed as soon as possible so that the resources are not unnecessarily busy. Additionally , network load should be well-balanced in order to provide a fair distribution of the resources used.

3.1.2 Design Options

The scheduling responsibility can be delegated on one centralized scheduler or be shared by multiple distributed schedulers. In this section we present the main reasons that lead us to the use of a decentralized solution.

The centralized approach is very simple, as there is only one scheduler for the Grid. A study performed by Zhang et al. (Zhang et al. 2003) shows that some centralized solutions such as Globus MDS fail to scale beyond 300 concurrent users e.g. the throughput begins to decline below acceptable levels. It has several important drawbacks that overcomes its use, such as having a single point of failure (Xhafa & Abraham 2010), lack of scalability (Dong & Akl 2006; Krauter et al. 2002; Xhafa & Abraham 2008; Xhafa & Abraham 2010) and lack of fault-tolerance (Dong & Akl 2006; Krauter et al. 2002; Xhafa & Abraham 2008).

Hierarchical solutions minimize the aforementioned problems by organizing the schedulers hierarchically. Keeping track of the hierarchy introduces additional complexity without completely solving the problems of centralized approach: it is more scalable and more reliable but in case of failure of a scheduler all the associated resources become unavailable.

In the decentralized scheduling algorithms there is no central scheduler to control the resources. Instead, there are local schedulers to which the scheduling requests are sent to. They take into consideration important issues such as fault-tolerance, scalability and multi-policy scheduling and so we used a decentralized scheduling architecture.

We present a decentralized architecture since it has more advantages when compared to centralized or hierarchical approaches. A decentralized scheduling architecture is harder to implement when compared to the other two approaches.

3.1.3 Grid Organization

Our decentralized architecture is based on the concept of VO, as depicted in Figure 3.1. Hence, the grid network is divided into different VOs, each one of them comprising three different types of entities:

- *Grid Information Service (GIS)* - that is used to maintain the information of the VO resources and also acts as a gateway.
- *Grid Resources* that are used for job's submission.
- *Scheduler (LS)* that participates in local and remote job scheduling.

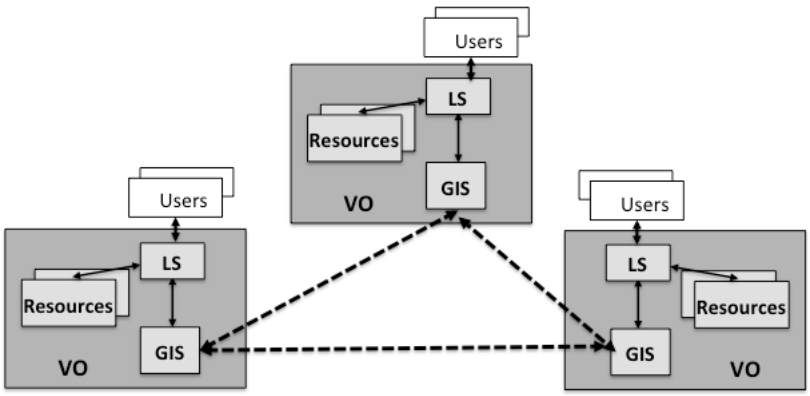

Figure 3.1: Grid organization

In our architecture, schedulers act as distributed entities that accepts jobs from local users or from other VOs and find the resource that best match the jobs' requirements, wherever they are located. Remote jobs are forwarded using the GIS. Hence, a job may be locally or remotely executed.

In spite of the importance of the resource distribution among resources and VOs organization, these topics are out of scope of this thesis and therefore they will not be addressed.

3.1.4 VO Architecture

An overview of a VO architecture is depicted in Figure 3.2, containing its main components and the interactions with other VOs.

As stated in the figure, the *Scheduler* comprises a *Resource Manager* and a *Job Scheduler*. The *Resource Manager* is responsible for maintaining a global view of the network and periodically transferring resource state with remote VOs so that each one of them maintains a snapshot of others VO resources'. The *Job Scheduler* is responsible for processing both local and remote jobs according to the scheduling algorithm in use, either by assigning them to a local resource or by forwarding it to a remote VO.

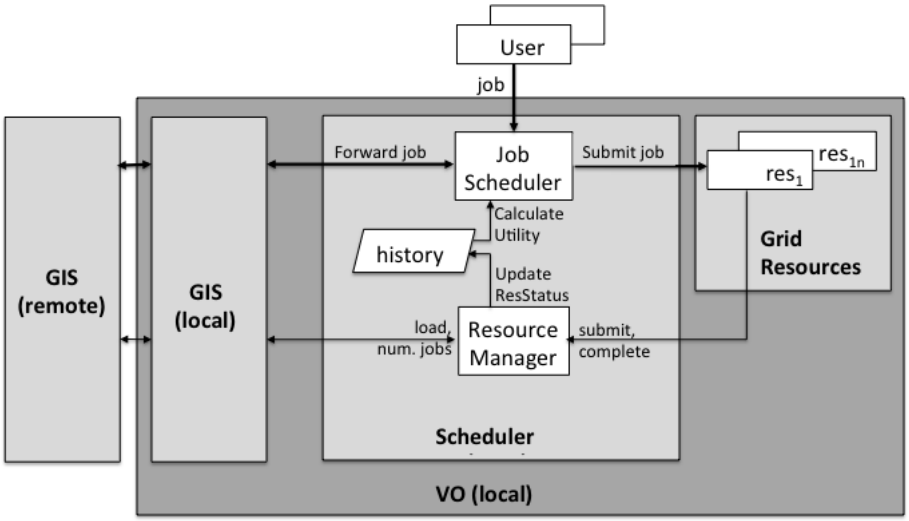

Figure 3.2: High Level Architecture

3.2 System modules

3.2.1 Grid Resources

From a network point of view, a Grid Resource is an end-system that is used to execute users' jobs. An accurate characterization of it should contain all the hardware and software characteristics, as well as the execution conditions. Hence, static attributes, such as architecture, type of processor, number of cores, hard-disk type and capacity, Random Access Memory (RAM), operating systems might be used. Dynamic attributes whose values change over time, like Central Processing Unit (CPU) occupation (per core), number of allocated jobs, free disk memory, free RAM and so on are also important to characterize the grid status.

A complete and detailed characterization of a resource will lead to a significant amount of information and complex scheduling decisions. Therefore, *Grid Resources* are characterized according to a set of properties that defines only the relevant attributes to the scheduling process. In our architecture a Grid Resource is modeled using both static and dynamic attributes. The following set of attributes have been considered:

- *Computer architecture* - that indicates a 32 bit or 64 bit computer.
- *Operating system* - that identifies the operating system used. Examples of possible values are:

Windows, GNU/Linux, MacOS, Solaris.

- *Number of cores* - that defines wether a single or multiple processor is used.
- *Processor speed* - that characterizes HW architecture in terms of Millions Instruction Per Second (MIPS) of each processor. MIPS is a raw measure of a computer's speed. MIPS does not reflect the exact speed of a computer since it does not take into consideration computer's Input/Ouput (I/O) performance, type of instructions or clock frequency. Despite these drawbacks, MIPS is a valid option since it has been widely used by the community and some companies to measure the cost of computing.
- Internal scheduling policy - that indicates the number of jobs that might be simultaneously assigned to each processor. In case of multiple jobs per resource one might need to take into account the way used by the operating system to deal with concurrency.
- *Resource Load* - that measures the amount of CPU that is allocated in each processor's resource in each time instant.

3.2.2 Scheduler

This section will describe one of the most important modules that were designed, the Scheduler. This module encapsulates all the behavior of the *Scheduler* that was developed in the context of this work. The rest of this section will describe each of the new entities that are part of this module.

3.2.3 Resource Manager

In order to have a snapshot of the entire set of the *Grid Resources*, each VO maintains local and remote resource status updated. The snapshot is stored in an hash table (*history*), which will be used for scheduling purposes.

Local resource status is kept up-to-date, as whenever a job is submitted or complete its execution, the resource snapshot is modified to cope with the new conditions. This procedure is executed for any type of job: local or remote.

Remote resources' status is updated upon reception of a remote resource snapshot that is periodically transmitted. In order to cope with the grid dynamics and have a more accurate view of remote resources, other sort of updates are needed. Hence, another mechanism used to maintain the status of remote VO is an explicit request for an update, when the local *Resource Manager* considers that its information regarding a given resource might be outdated. The information is marked as outdated if either the load or the number of submitted jobs exceeds the defined threshold values that were assigned to

each one. The counters used to monitor the number of jobs that satisfies these conditions are reset when an update of the resource information is made. There are two counters: one for the load estimation of a resource and another for the number of submitted jobs without refresh. When these counters achieve a threshold value, a refresh request is sent by the LS.

The pseudo-code of the *Resource Manager* is depicted in Algorithm 1.

Algorithm 1 Resource Manager algorithm

```
WaitFor(event)
if event == (Submit (job, res) OR Complete (job, res)) then
   LocalUpdate (history(local_VO, res))
end if
if event==(Timeout) then
   SendUpdate (history (to_all_VO, all_local_res))
end if
if event==(JobLimit  OR  LoadEstimation) then
   SendRequest (history(to_VO, to_res))
end if
if event == ReceiveUpdate (history (from_VO, any_res)) then
   LocalUpdate (history(from_VO, any_res))
end if
if event == ReceiveRequest (history(from_VO, from_res)) then
   SendUpdate (history(to_VO, to_res))
end if
```

3.2.3.1 Load Estimation

Resource load estimation allows the *Scheduler* to have a more consistent view of other VO's resources' state. The procedure is triggered by the *Job Scheduler,* , upon forwarding of a job to a remote VO. Based on the information of the job's requirements, the *Resource Manager* predicts the load of that resource, considering that the job will be accepted by the remote VO.

The load of a resource is the average load of all its Processor Elementss (PEs) as it is presented on Equation 3.1.

$$Resource\_Load = \frac{\sum_{i=1}^{N} load(PE[i])}{N} \tag{3.1}$$

Where N = number of PEs and,

$$Load(PE) = \frac{total\_mips - free\_mips}{total\_mips} \tag{3.2}$$

3.2.3.2 History

Each scheduler has an hash table, called *history*, that contains a snapshot of other VO's resources. Like every hash table, *history* is formed by a set of key-value pairs. The key set is composed by all the ID's of grid's GIS. For each key, the value set is a list that contains information about each GIS's resource called *history entry*. Each entry comprises:

- *Resource characteristics* - list of static attributes used to model a resource. In our case, it comprises the computer architecture, operating system, number of cores and processor speed.
- *Update timestamp* - instant of time when the last update of the entry was made;
- *Submitted jobs* - information about submitted jobs. It contains the job id, utility value, execution time and final status (failed or success);
- *Job counter* - information about the number of jobs that have been submitted during the current update cycle;
- *Resource load* - estimation of the load of the resource;
- *Number of running jobs* - number of jobs that are running on the resource.

3.2.4 Job Scheduler

From an high-level perspective, the *Job Scheduler,* is a very simple algorithm, which basically performs two different actions, depending on wether the job being submitted is local or remote: In case of local job, it executes a *Local Job Submission* procedure that selects the most adequate resource, if available, and processes the job accordingly. In case of remote job, it executes a *Remote Job Submission* procedure that verifies if the required resources are available and processes the job accordingly. In both cases jobs may be locally submitted, forwarded to a remote VO or rejected.

The pseudo-code for the described procedure is presented in Algorithm 2.

Algorithm 2 Job Scheduler algorithm

```
while TRUE do
  job = WaitForJobSubmission()
  if job ∈ remoteVO then
    RemoteJobSubmission(job)
  else
    LocalJobSubmission(resource,job)
  end if
end while
```

3.2.4.1 Resource selection

The main function of the *Scheduler* is to find a resource to allocate a job either from a local or a remote user. This section will describe the steps that are taken in order to select a resource for a specific job.

The *Resource Selection* procedure has two important inputs: the job and a set of resources. The first step is to filter the resources that are available. A resource is considered unavailable if all its PE's are busy or the estimated execution time exceeds the limits imposed by the user. By doing this analysis to all resources in the input set, the *Scheduler* builds a subset that contains all the potential candidates for selection. If the subset is empty it means that no potential candidate resource was found and the process ends. Otherwise, the *Scheduler* calculates the maximum utility value of each resource in the subset using our new algorithm. If the maximum utility value is zero that means that none of the resources satisfies the job's requirements. The process ends here. If the maximum utility value is greater than zero the *Scheduler* selects the resource that maximizes the utility. If multiple resources match the criteria, the one having the lowest load is selected. When the resource is selected the process ends.

The pseudo-code for the described procedure are presented in Algorithms 3 and 4.

Algorithm 3 Resource Selection algorithm

```
res_list = FilterResources(history(local_VO, all_res, job))
while res ∈ res_list do
  if res != BUSY then
    AddToList (available_res_list, res)
  end if
  res = NextResource (sel_list)
end while
if available_res_list ∈ NULL then
  res = NO_RESOURCE_AVAILABLE
else
  utility = CalculateMaxUtility (available_res_list)
  if utility == 0 then
    res = NO_RESOURCE_AVAILABLE
  else
    res = SelectResource (MaxUtil, LowestLoad)
  end if
end if
return (res)
```

Algorithm 4 Calculate Max Utility algorithm

```
for all res ∈available_res_list do
  res_util = CalculateUtility (job, res)
  AddToList (util_list , res_util)
end for
return MaxUtility( res_util)
```

3.2.4.2 Local Job Submission

This section describes all the steps that the *Scheduler* takes when it receives a local job submission request, e.g. a job request from a user from the same VO.

The first step is to find a resource to submit the job by using the *ResourceSelection* procedure. If no potential candidate is found, the job is declared as failed and the procedure returns. Otherwise the *history entry* of the selected resource is updated by the *Resource Manager*. After that, it is checked if the selected resource is local or is from another VO. If the resource is local, then the job is sent for submission. Otherwise, the job is forwarded to the GIS of the selected resource. In order to avoid keeping forwarding jobs among VOs indefinitely, a Time-To-Live (TTL) value is defined.

The pseudo-code for the described procedure is presented in Algorithm 5.

Algorithm 5 Local Job Submission algorithm

```
sel_res = ResourceSelection(job, history(local_VO, all_res))
if sel_res == NO_RESOURCE_AVAILABLE then
   return (Job Failed)
end if
if sel_res ∈ localVO then
   Submit (job, res)
   return (Submit)
else
   SetJobTTL (job, MAX_TTL)
   Forward (job, to_VO_res)
   return (Forward)
end if
```

3.2.4.3 Remote job submission

Besides making local job submissions, a *Scheduler* may receive jobs that were forwarded by other schedulers. This section describes the actions taken by a *Scheduler* when receiving a remote job.

When a remote job arrives, the *Scheduler* checks wether the required resource is available or not. If the resource is available, the job is submitted and the *history entry* of the selected resource is updated by the *Resource Manager*. Otherwise, it checks if it has other local available resources to submit the job, as another forwarding will increase the latency. If there is any local available resource the job is submitted and the *history entry* updated as well. If this is not the case, the job may be either forwarded to another VO, if the TTL is valid, or reject and declared as failed, if no more hops are allowed.

The pseudo-code for the described procedure is presented in Algorithm 6.

Algorithm 6 Remote Job Submission algorithm

```
if res != NOT_BUSY then
   Submit (job, res)
   return (Submit)
end if
sel_res = ResourceSelection(job, history(local_VO, all_res))
if sel_res == RESOURCE_AVAILABLE then
   Submit (job, res)
   return (Submit)
else
   TTL = GetJobTTL (job)
   Decrement (TTL)
   if TTL == NULL then
      return (Job Failed)
   end if
   Forward (job, to_VO_res)
   return (Forward)
end if
```

3.3 Partial Utility scheduling

Next, we will describe how we calculate our utility function. Our solution is based on the work of Silva et al. (Silva et al. 2010) that was proposed for a peer-to-peer network. Unlike other schedulers that select a resource that is able to satisfy all the user requirements (Matchmaking mechanism (Raman et al. 2000; Thain et al. 2003)) we propose a more flexible approach on requirement fulfillment by introducing the notion of partial utility or partial requirement fulfillment.

3.3.1 Job's requirements

In our architecture, users may submit jobs and specify the list of different requirements that the resources must satisfy to be used. Hence, architecture, operation system, number of cores, as well as a maximum time for job completion might be defined during the job creation.

In order to cope with the variety of resources' options that are available in a VO, users may also specify how important is each one of the available options and rank them according to their preferences. For instance, a user may prefer an MacOS resource, but if it is busy he would rather used a Linux machine instead of using a Windows-based system. Hence, the user can specify more than one option per requirement with the corresponding utility value (ranging between [0,1]). The different utility values that are assigned to each requirement's option define the *Partial Utility*.

3.3.2 Partial Utility function

A key component of the *Scheduler* architecture is the algorithm used to select the resource allocated to each one of the jobs, the *Utility Function*. This section describes our utility function, the Partial Utility (PU).

To select the most adequate resource for each job, the *Scheduler* needs to match the job's requirements with the attributes of the available resources. The global utility value weights the different requirements equally and the final result is a combined aggregation of the combined satisfaction of requirements. For each resource, either local or remote, its utility is calculated according to the resource status information and the job's requirements. The best resource is the one that maximizes the utility function.

Let us now define how the utility of each resource, res_util, is calculated.

Considering:

- A list of job requirements: $jobReq = \{req_1, ...req_N\}$;
- For a given requirement, req_i, the list of possible options: $opt_i = \{opt_{(i,1)}, ..., opt_{(i,K)}\}$;
- The set of utility values of requirement's (req_i) options (opt_i): $\alpha(i)$
- The weight $\beta_i \in [0, 1]$ assigned to each requirement req_i.

The res_util of resource res is given by Eq. 3.3:

$$res\_util = \frac{\sum_{i=1}^{N} max(\alpha_{(i)} * \beta(i))}{N} \tag{3.3}$$

If all the requirements have a similar importance β must be set to one. However, if one wants to prioritize them, a different value of β must be assigned to each one of them.

3.4 Alternative scheduling algorihms

In order to evaluate the performance of our solution we need to compare it with other alternative scheduling algorithms. This section describes the other selected algorithms, namely the Round Robin (RR), the Binary Utility (BU) and the Matchmaking (MM) .

3.4.1 Round Robin

The RR (Tanenbaum 2007) algorithm is one of the most used scheduling algorithms. In RR there is only one scheduler that is responsible for processing all jobs and contains a list with information about all resources. The scheduler also has an index that is refers to the actual position in the list. Every time a job arrives, the scheduler selects from the resource whose position is equal to the index value. The utility value is calculated and the job is sent to that resource without checking if there is another resource that maximizes utility. After the job is sent to submission the index value is incremented. Due to its nature, the RR algorithm only works on centralized scheduling architectures.

3.4.2 Binary Utility

This algorithm treats each requirement utility calculation in a binary way. If the resource satisfies the given requirement than maximum utility is granted. Otherwise, that requirement's utility is zero. In order to achieve this behavior each job's requirement has only one option, which has the maximum utility value, e.g. the value 1. The resource utility is calculated in the same way that is used by the PU algorithm.

Let us now define how the res_util of each resource is calculated in the BU algorithm.

Considering:

- A list of job requirements: $jobReq = \{req_1, ...req_N\}$;
- For a given requirement, req_i, there is only one option: opt_i;
- The utility value of that option: $\alpha_{(i)} = 1$
- A parameter, $\gamma_i \in \{0, 1\}$ that defines if the requirement (req_i) was fulfilled or not

The res_util of resource res is given by Equation 3.4:

$$res\_util = \frac{\sum_{i=1}^{N} \alpha_{(i)} * \gamma_{(i)}}{N} \tag{3.4}$$

3.4.3 Matchmaking

The Matchmaking algorithm can be seen as a more restricted variation of the BU algorithm. Like BU, in MM there is only one option per requirement which has maximum utility if the resource satisfies the

requirement or zero otherwise. The difference between MM and BU relies in the way the global utility value is calculated. In the MM algorithm the global utility is also binary, e.g. one or zero.

Let us define how the res_util of each resource is calculated in the MM algorithm.

Considering:

- A list of job requirements: $jobReq = \{req_1, ...req_N\}$;
- For a given requirement, req_i, there is only one option: opt_i;
- The utility value of the requirement : $\alpha_{(i)} = 1$
- A parameter, $\gamma_i \in \{0, 1\}$ that defines if the requirement (req_i) was fulfilled or not

In the MM algorithm, the res_util of resource res is given by Equation 3.5:

$$res\_util = \frac{\Pi_{i=1}^{N} \alpha_{(i)} * \gamma_{(i)}}{N} \tag{3.5}$$

Equation 3.5 shows that the final utility of a resource, for a specific job, is one only and if only all the job's requirements are fulfilled. In other words, the resource is considered suitable to execute the job only if it fulfills all its requirements. MM can be seen as more restricted version of the BU algorithm. The MM algorithm is based on Condor-G's Matchmaking mechanism (Imamagic et al. 2006b).

3.5 Synthesis and Final Remarks

This chapter describes our decentralized scheduling architecture. In the proposed solution resources are grouped into VO, each one of them having its own *Scheduler*. To determine the most adequate resource a *Resource Manager* monitors the status of the resource information and stores it in an hash table, the *history* table. The *Job Scheduler* is responsible for the scheduling decision, and, using their knowledge of all VOs decides if a particular job must be locally submitted or forwarded to a remote VO.

During job creation, users may define a set of requirements and rank the possible options of each one of them, by defining their *Partial Utility*. Jobs may be submit either to local or remote resources, depending on the resources that maximizes the utility functions and guarantees job completion time and resource lowest load. This function equally weights the maximum utility value of each requirement.

The use of a decentralized architecture is a key aspect to provide a scalable solution, as a large number of resources and users' jobs can be distributed among different schedulers. However, the number of VOs and the number of resources of each VO constrains the degree of scalability that might be achieved,

and these topics are out of the scope of this thesis. Nevertheless, simulation studies will illustrate the impact of the variation of these parameters.

Although Grid's architecture restricts the failure problem of a *Scheduler* to the local jobs of the VO, no reorganization of resources is made in the present version of the architecture. Hence, limited reliability is achieved.

The flexible job requirements' definition and resource attributes, combined with the use of an utility function based on the partial fulfillment of job's requirements lead us to anticipate that multi-policy scheduling, user satisfaction and performance requirements are met. However, simulations studies are needed to proof it.

4 Implementation

This chapter deals with all the topics related to the implementation of the solution that was proposed in chapter 3. Important aspects will be analyzed and described such as the work tool, the modules and processes that were developed and changes that were made to the original tool due to the solution's requirements. The chapter is organized as follows. Section 4.1 describes the simulator modular architecture and some of its most important modules in detail. Section 4.2 details the extensions that were made to the simulator, namely modifications to existing classes and addition of new ones. The methodology used to implement the grid network topology is described in section 4.3. Section 4.4 summarizes the chapter and presents some final remarks.

4.1 The GridSim simulator

Many grid simulators have been presented in Section 2.2. The chosen simulator for this work was GridSim (Buyya & Murshed 2002). GridSim allows the creation of network topologies, resources with extendable allocation policies, provides a GIS entity and it is the most used simulator by the community (see table 2.2 for more details). The quality of the code's documentation and set of examples was also an important factor of decision.

This section will describe the most important components of GridSim. GridSim provides a comprehensive facility for simulation of different classes of heterogeneous resources, users and schedulers. GridSim can be used to simulate application schedulers for distributed computing systems such as clusters and grids. Some of the features included in GridSim are: resource modeling operating under space or time-shared mode, resource capacity can be defined, no limit of jobs that can be submitted to a resource and resources can be located in any time zone, just to name a few.

GridSim is developed in a modular way that allows developers to extend it and implement new features such as new scheduling policies, for example. The modular architecture is presented in Figure 4.1. In the rest of this section two important modules will be described: basic discrete event simulation infrastructure and the GridSim toolkit.

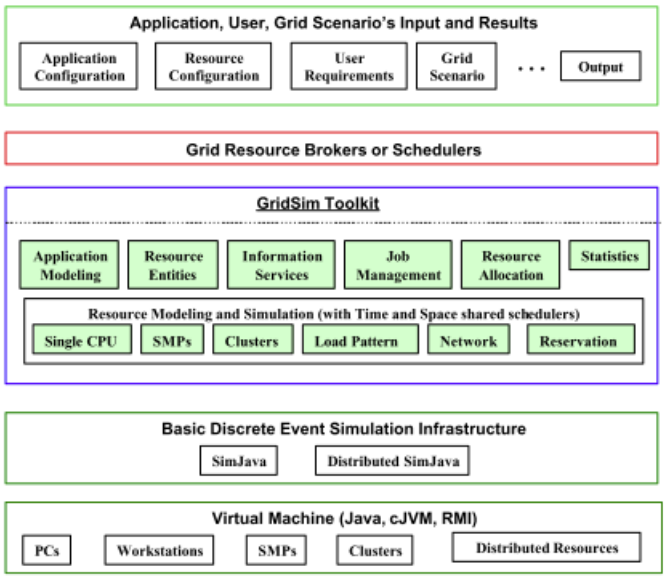

Figure 4.1: GridSim modular architecture

4.1.1 Discrete event simulation

GridSim uses a discrete event simulation package called SimJava (Howell & Mcnab 1998). In SimJava, a simulation contains a number of entities each of which runs in parallel on its own thread. The entity's behavior is encoded in java using its *body()* method. Each entity has access to a small number of primitives. These primitives enable the construction of a network of active entities that communicate by sending and receiving passive event objects.

- *sim_schedule()*: sends event objects to other entities via ports;
- *sim_hold()*: holds for some simulation time;
- *sim_wait()*: waits for an event object to arrive;
- *sim_select()*: selects events from the deferred queue.

In SimJava, event objects are passed to another entity via ports using *sim_schedule()*. They are automatically queued, and retrieved as required by the receiver using *sim_select()* and *sim_wait()*. The *sim_select()* primitive is used to select from events that have already arrived. *Sim_wait()* waits for the next future event. This is called the entity interaction model. All events are globally sorted by simulation timestamp to ensure that messages never arrive out of order. The basic sequential discrete event simulation algorithm is as follows. A central object *Sim_system* maintains a timestamp ordered queue of future events. When the simulation begins, all entities are created and their *body()* method is run. When an entity calls a function, lets say *sim_hold(10.0)*, the *Sim_system* object halts that entity's thread

and places an event on the future queue meaning that the hold will complete at $sim\_time + 10.0$. When all entities have halted, *Sim_system* pops the next event off the queue, advances the simulation time accordingly and restarts entities as appropriate. This procedure will continue until no more events are generated. If the Java Virtual Machine (JVM) supports native threads, then all entities starting at exactly the same simulation time may run concurrently.

4.1.2 GridSim Toolkit

The GridSim Toolkit is the core of GridSim. It is built on top of SimJava and adds specific features related to grid scenarios. The rest of this section describes the core features that are implemented by GridSim.

GridSim supports entities for simulation of single processor and multiprocessor, heterogeneous resources that can be configured as time or space shared systems. It allows setting their clock to different time zones to simulate geographic distribution of resources. It supports entities that simulate networks used for communication among resources. During simulation, GridSim creates a number of multi-threaded entities, each of which runs in parallel in its own thread. An entity's behavior is simulated within its *body()* method, following the same approach as SimJava. In order to address some grid issues, some more specific entities were developed by the GridSim's authors.

4.1.3 User

The *User* entity represents a grid user. This entity is responsible for creating and sending jobs for submission. It has methods to communicate with the GIS to get information about all the resource in the grid.

4.1.4 GridResource

One of the most important entities that were added, comparing with SimJava, is the *GridResource* entity. A physical resource is modeled by two different set of attributes.The first one characterizes the resource in terms of computer architecture, operating system and internal scheduling policy whist the second one comprises a list of machines that characterizes the processors' cores. This characterization comprises the definition of the processor core, given by PE; and the definition of the processor speed, given by MIPS.

Every *GridResource* entity has an internal scheduling policy, *AllocPolicy*. The policy is responsible for managing all job's allocation and processes on the resource's PEs. The internal policy can pause, cancel, resume and submit jobs as well moving jobs to another resource and also maintains information about the jobs that are in execution, paused or waiting. The *AllocPolicy* entity is responsible for executing the operations related to the resource's progress in the simulation, i.e. schedule internal events related to

information obtained from the jobs that are in execution. The internal event's time interval is as large as the time required for the completion of a smallest remaining-time job. GridSim implements two different internal scheduling policies: time and space shared.

In the *Time Shared Policy* all jobs share PE's MIPS and all PEs have the same MIPS capacity. The first N jobs on the execution queue receive more MIPS than the others. The variable N is calculated using equation 4.1 where, J is the number of jobs in execution and P is the number of PEs of the resource. The maximum MIPS allocated to the first N jobs is calculated using equation 4.2. In equation 4.2, M is the number of MIPS per PE and T is the *timespan* (in seconds). This policy does not consider job's size but the number of jobs in execution when allocating the number of MIPS . Each PE can process multiple jobs and it is represented in Figure 4.2.

$$N = P - (J \bmod P) \times \frac{J}{P} \qquad (4.1) \qquad Max\_mips = (M \times T) \times \frac{P}{J} \qquad (4.2)$$

In the *Space Shared Policy*, each PE can only run one job at a time as it is depicted in figure 4.3. When a job arrives for submission, the policy checks if there is any free PE. If there is, at least, one PE available, the job is submitted and the selected PE becomes busy. Otherwise, the job is put on the waiting queue. When a job finishes the policy frees the PE allocated to it and checks if there are any queued jobs. If there are queued jobs, the policy selects the first one and assigns it to the PE, which is free.

The *AllocPolicy* entity is build in a way that is possible for developers to extend it by adding new features and behaviors.

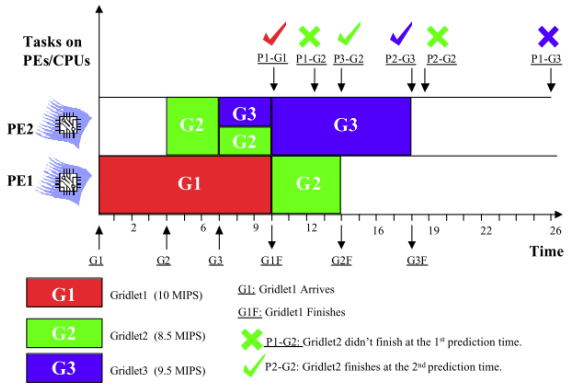

Figure 4.2: Time shared

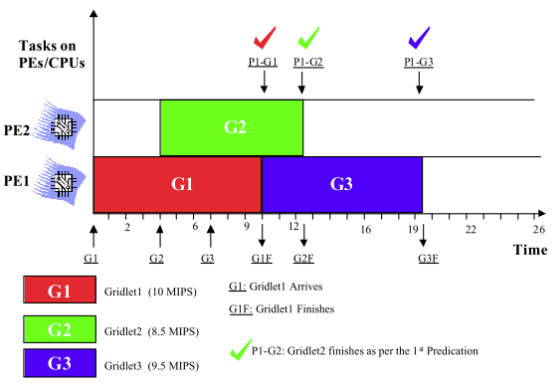

Figure 4.3: Space shared

4.1.5 Grid Information Service

The *GIS* entity is responsible for managing a list of resource available in the Grid. The GIS also provides registration services to grid resources. The *GIS* entity is built in a way that is possible for developers to extend it by adding new features and behaviors.

4.1.6 Input and Output

The flow of information among the GridSim entities happen via their I/O entities. Every networked GridSim entity has I/O channels or ports, which are used for establishing a link between the entity and its own I/O entities. I/O entities have their own execution thread with *body()* method that handles events. The use of separate entities for input and output enables a networked entity to model full duplex and multi-user parallel communications. The support for buffered input and output channels associated with every GridSim entity provides a simple mechanism for an entity to communicate with other entities. Figure 4.4 depicts the communication model between I/O entities.

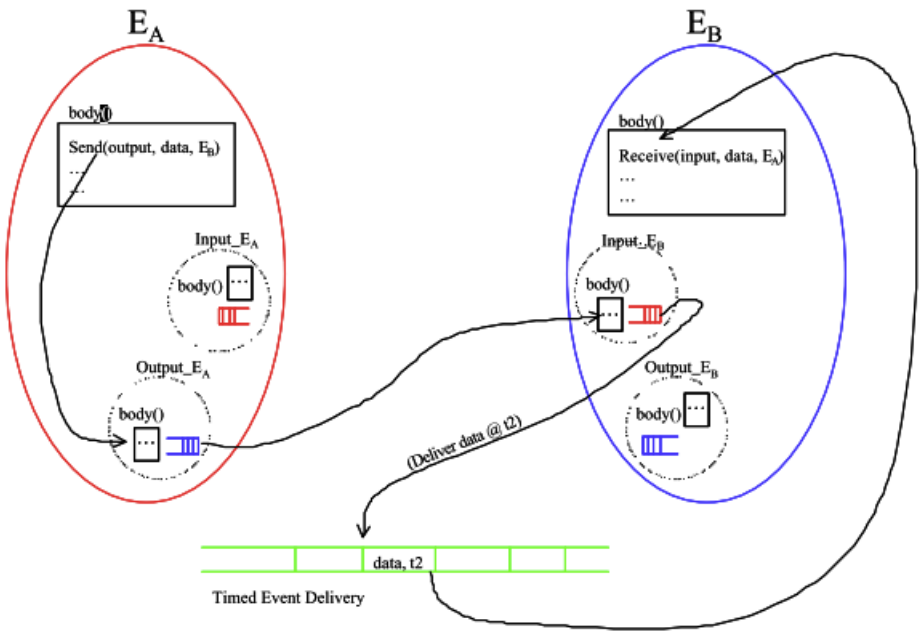

Figure 4.4: I/O entities communication model

4.1.7 Gridlet

The *Gridlet* entity contains all information related to a job and its execution management details such as its length (in Million Instructions (MI)), the size of input and output files, number of PEs required and creator (typically is a grid user). These basic parameters help in determining execution time, the

time required to transport input and output files between users and remote resources, and returning the processed jobs back to the originator along with the results.

Each *Gridlet* entity maintains information about its current state. These states can be divided into two categories: process and final. Process states are related to the current situation of the *Gridlet* when it has not terminated. The *Gridlet* entity process states are: ready, queued, in execution, paused and resumed. The states that are in the final category are: success, failed, canceled. Table 4.1 presents a taxonomy of the states that were previously mentioned.

States	
Process	Ready Queued In Execution Paused Resumed
Final	Success Failed Canceled

Table 4.1: Gridlet's states

4.1.8 Interaction protocols model

The protocols for interaction between GridSim entities are implemented using events. The events can be raised by any entity to be delivered immediately, or with specified delay to other entities or itself. The events that are originated from the same entity are called *internal events* and those originated from the external entities are called *external events*. Entities can distinguish these events based on the source identification associated with them. Events can be further classified into synchronous and asynchronous events. An event is called synchronous when the event source entity waits until the event destination entity performs all the actions associated with the event (i.e., the delivery of full service). An event is called asynchronous when the event source entity raises an event and continues with other activities without waiting for its completion. When the destination entity receives such events or service requests, it responds back with results by sending one or more events. It should be noted that external events could be synchronous or asynchronous, but internal events need to be raised as asynchronous events to avoid deadlocks. Table 4.2 summarizes the event's classification that was made.

Events	
Origin	Internal External
Protocols	Synchronous Asynchronous

Table 4.2: Types of events

4.2 Extensions to the Simulator

Due to the nature of this work, several extensions were added to GridSim. This section presents and describes the additions that were developed and is organized as follows: first the new modules are described, next there is a description about new functionalities of some GridSim entities (*RegionalGIS*, *GridResource*, PE) as well as the presentation of a new message type that is used in all communications between entities.

Figure 4.5 depicts the modular functional architecture of GridSim that comprises twelve modules. The modules whose background is white have not been modified. The ones that have a yellow background have been modified and the modules that were developed from scratch are identified by having a green background.

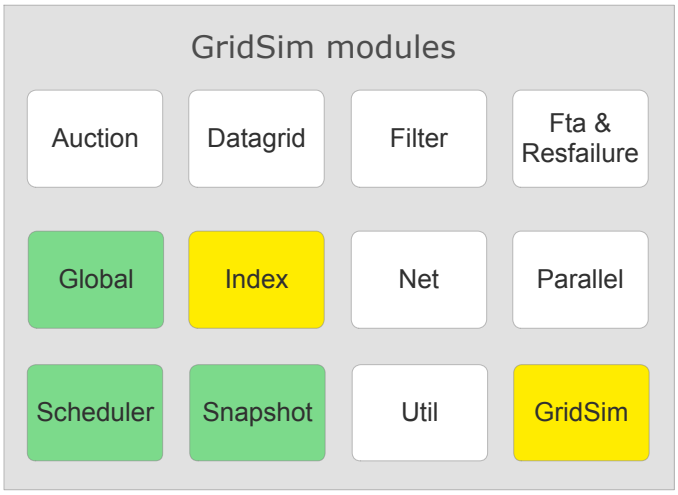

Figure 4.5: GridSim functional modules

The *Auction* module provides a framework for the auction model. This model is used mostly by economic-based market solutions to scheduling such as the work of Chunlin et al (Chunlin & Layuan 2005). The *Datagrid* module provides a framework for datagrid models. This module allows the creation of jobs that require specific data files to run as well as resources with storage capacities and replicas. The *Filter* module provides a framework to filter jobs and other events that match some given requirements (transaction id and tag name). The *Fta* and *ResFailure* modules are very similar because they add failure functionalities to the simulator. The *ResFailure* module provides a framework for resource failure model module and the *Fta* module provides a framework to simulate failures wether their are events, resources.

The *Global* module was developed due to some solution's requirements. It contains all global variables such as simulation setup time, link's capacity in $bytes/s$ and the seed of the random number generator. The module has information about the locations of important files and folders such as logs

and results. This module also has a method that calculates a size, in bytes, of a given object. This functionality is important for sending objects between entities using I/O ports. The *Index* module provided a framework for GIS models. This module simulates the behavior of multiple regional GIS by performing basic functionalities such as storing a list of local resources. Due to the nature of this work, this module was extended to address some important needs such as resource refresh requests and job forwards, for example. A detailed description of the GIS' new features can be found in section 4.2.2.2.

The *Net* module provides a framework for building network topologies with GridSim's entities. This module allows the creation of routers and links to connect different entities where the Maximum Transmission Unit (MTU) and capacity can be specified. The *Parallel* module provides a framework for creating resources with concurrent entity capabilities. This module includes extensions to the standard GridSim's GIS and allocation policies entities that add failure functionalities.

The *Scheduler* module was developed from scratch due solution's requirements. This module is responsible for all the processes associated with the schedulers. A very detailed description about this module can be found on section 4.2.1. The *Scheduler* module responsible for selecting a (local or remote) resource for a specific job. Besides resource selection and job submission, this module also contains a very important set of entities such as: the *History* entity that is responsible for keeping snapshots of the state of other VOs, the *JobRequirements* entity that contains all information about a job's requirement, the *SchedulingInformation* entity that encapsulates the scheduling data of a job and an entity that is responsible for handling the resources' internal allocation policy, *UtilityAllocationPolicy*. The *Util* module provides some statistical functionality such as the Poisson and HyperExponential distributions.

The *GridSim* module contains much of the most important entities that represent some of the most important features in the simulator such as event tags, jobs, resources, default allocations policies, PEs, etc. This module was extended to address some solution's requirements. Cloning mechanisms were introduced in *ResourceCharacteristics*, *Machine* and *PE* entities a set of new features were added to the *Gridlet* entity such as a set of requirements, scheduling information, forwarding TTL and maximum execution time, just to name a few.

4.2.1 Scheduler module

This section will describe one of the most important modules that were developed, the *Scheduler* module. This module encapsulates all the behavior of the scheduler that was developed in the context of this work. The rest of this section will describe each of the new entities that build this module and is organized as follows. Section 4.2.1.1 will describe the *Scheduler* entity and the processes related to it such as its finite state machine, Section 4.2.1.2. Section 4.2.1.3 will describe the important aspects of the *JobRequirements* entity such as its attributes, how different utility options are represented and how requisites

and utility intervals are generated. Section 4.2.1.4 will describe the *SchedulingInformation* entity's attributes and its importance in the module. Finally, section 4.2.1.5 will describe the *UtilityAllocationPolicy* entity and how it manages the internal events of grid resources.

4.2.1.1 Scheduler

The *Scheduler*, the major addition to GridSim, is the core of this work since it is responsible for selecting resources for jobs according to different scheduling policies and requirements. This section will describe some important aspects about the *Scheduler* such as: states, local job submission procedures, remote job submission procedures, resource selection and monitorization. The *Scheduler* is an extension of the *GridSim* entity. The *GridSim* is an extension of the *GridSimCore* entity. The *GridSimCore* entity is the main core of the all the GridSim's entities. This entity performs event and I/O management, has the capacity to send objects to other entities (with or without tags), can send ping requests to other entities and can generate background traffic. The *GridSim* entity performs actions related to job management such as submitting a job to a resource (given it's IDentification (ID)) with a given delay, cancel a job that is running on a resource, move a job from a resource to another, pause a running job during a given period of time and resume a job that is currently paused. Besides job management, the *GridSim* entity can also send requests to it's regional GIS, get information from a resource such as its characteristics and setup a new GIS. The feature of adding a new GIS is useful to when the new GIS has some specific behavior.

4.2.1.2 Finite State Machine

The *Scheduler* has four different states: *History Update I*, *History Update II*, *Normal* and *End*. The first two states, *History Updated I* and *History Updated II* can be seen as a setup phase before the job scheduling simulation.

When a *Scheduler* entity starts it goes to *History Update I* state. At this point, the *Scheduler* only knows its GIS but, in order to have a view of other VO's state it needs to know who are the other VO's. The *Scheduler* sends a request to its GIS and waits until it receives a response containing all the GIS ids on the grid. All other events are not processed, and are sent into a queue.

When the response is received the *Scheduler* transitions to state *History Update II*. On this state, the *Scheduler* needs to get information about resources to build its view of Grid (History). On GridSim, resources need to register themselves to their regional GIS. When all resources are registered, the GIS sends information about its resources to its local scheduler and to all other GIS on the Grid. When a GIS receives information about other GIS's resources, it forwards that information to its local scheduler. The scheduler builds a table that associates each GIS ID with the set of resources received.

The *Scheduler* will move to *Normal* state when it has received information from all the GIS on the grid. The time from the beginning of the simulation until the scheduler reaches the Normal state is called *simulation setup time*. During simulation setup time, users cannot send any jobs for submission. On *Normal* state the *Scheduler* will process every simulation event. Some important event tags are:

- *LOCAL_JOB_SUBMIT:* event related a local job submission;
- *REMOTE_JOB_SUBMIT:* event related to a job that was forwarded to another scheduler;
- *REMOTE_JOB_RESULTS:* event related to the receiving of results of a remote submission;
- *GRIDLET_RETURN:* event related to the receiving of results of a local submission.

Simulation will end when the scheduler receives the End of Simulation event and transit to the *End* state. On the End state the scheduler will shutdown all I/O ports and exit. Figure 4.6 represents the finite state machine diagram of the *Scheduler*.

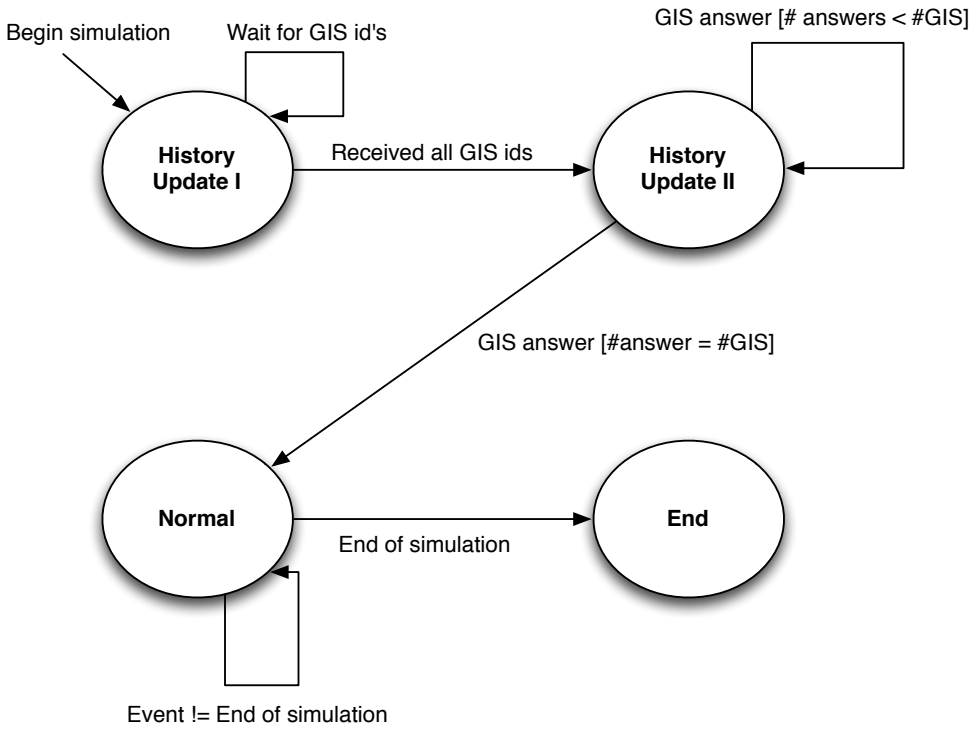

Figure 4.6: Scheduler's States

4.2.1.3 Job Requirements

In the proposed solution, users may submit jobs that have different requisites, each one of them with a set of possible options. In GridSim, the concept of job requirement did not exist. In all solutions

that were analyzed (see Sections 2.4.3 and 2.4.4) there was no concept of job requirement. The length was the only characteristic associated with a job. Also, the simulation environment was very simple because it did not consider a realistic grid scenario but only a set of resources and a queue of jobs. In order to describe jobs in a more realistic way it was necessary to create a class to represent the different requirements each job could had. That class is called *JobRequirements*. A list of the requirements that are supported by this class is presented below:

- *Operating system:* represents the various types of possible operating systems (Linux, Mac OS, Windows, Solaris)
- *Architecture:* represents the various types of possible architectures (32 bits, 64 bits);
- *Execution time:* a number that represents the desired execution time for the job (double > 0);
- *Number of cores:* a number that represents the number of cores that the machine needs to have in order to submit the job (integer > 0);
- *Processor speed:* the processor speed desired to execute the job (double > 0)

This class was integrated in the *Gridlet* class (class that represents the job in GridSim) using an hash table. The keys are the names of the requirements and the values are the job's requirements, i.e. the *Job Requirements* class. Due to the nature of the proposed solution, each requirement can have more than one option (with different utility values).

Requirements can be classified in two ways: numerical and non-numerical. Numerical requirements are those whose options are numbers between a well defined interval. Finish Time, Number of Cores and Processor Speed are numerical requirements. Non-numerical requirements have no numerical representation. Operating system and Architecture are non-numerical requirements. Each *JobRequirement* class has a list on which each element contains information about the different possibilities for that requirement (Utility Interval Value).

The Utility Interval Value class contains four very important attributes:

- *Utility value:* number between 0 and 1 that represents the level of satisfaction of the user related to this option;
- *Lower limit:* lowest value that the option can take in order to have the option's utility value (only applicable to numerical requirements);
- *Upper limit:* highest value that the option can take in order to have the option's utility value (only applicable to numerical requirements);

- *Requirement value:* only applicable to non-numerical requirements. Represent the different kinds of operating systems or architectures.

4.2.1.4 Scheduling Information

When the scheduler calculates the utility value of a job on a specific resource the result is important information associated with that process that must be saved. The scheduler needs to know the IDs of the machine and the PE that were selected as well as the MIPS that the job requires. Besides that, the scheduler also needs to know if the resource is local or remote and the expected execution time value. The scheduler must know if the resource cannot process the job because all its PEs were busy. Finally, it is important to know the calculated utility value of each requirement. An example on how important is to have information about each requirement's utility value is when a resource receives a job for submission and the desired PE is busy. This situation is described in section 4.2.1.5. In order to encapsulate all this information a class, named *SchedulingInformation*, was created.

The *SchedulingInformation* class contains the following important attributes:

- *Utility value*: the final utility value of the resource. This value is a the average sum of the utility values of all requirements but could easily be weighted with different weights per requirement using other metrics such as geometric averages;
- *Remote resource*: a boolean that indicates if the resource is remote or local. This checking is performed by checking the resource's GIS ID with the GIS ID of the scheduler;
- *Expected execution time*: a double greater than zero that represents the expected execution time of the job on the resource;
- *Required MIPS*: the capacity that the job must use in order to fulfill the time requirements. A variation in value has direct influence in the execution time and finish time utility values;
- *is PE busy*: a boolean that indicates if all the resources PE are busy or not. This attribute is crucial to determine if the resource is available or not for execution. A situation can occur when a resource is busy (all its PE are busy) and a job has a global utility value greater than zero. This can happen if the resource satisfies operating system and architecture requirements, for example. However, despite the final utility value is greater than zero, the resource cannot be considered, otherwise the execution time of the job could not be estimated.
- *Requirements utility info*: this attribute is a private class of Scheduling Information and it is responsible for storing information regarding each requirement's utility. As so, it has five attributes that represent the utility values of all requirements (operating system, architecture, execution time,

number of cores and processor's speed). It is this class that is responsible for computing the final utility value by doing an average sum of all requirement's utility values. Separating the utility values of all requirements allows each one of them to be recalculated individually without the need to recompute the other values.

This class is used in many steps during the scheduler' resource selection procedure. Selecting the resource with the lowest load among the resources with the highest utility value, checking if a resource is busy or not, forwarding a job to another GIS or submitting it locally by checking if the selected resource is remote are examples of were information from Scheduling Information is used.

4.2.1.5 Utility Allocation Policy

GridSim has a class, called *AllocPolicy*, that handles the internal resource allocation policy. This class is extendable so each developer can implement its own internal resource allocation policy. Two types of policies were already implemented: Time shared and Space shared.

None of two implemented policies satisfied the requirements that the proposed solution needed, so a new policy, called Utility Allocation Policy (UAP), was developed. In this new policy, a PE can process multiple jobs There are four important states in UAP: update job processing, process job submission, check and process job completion, make periodic snapshots of the resource's state.

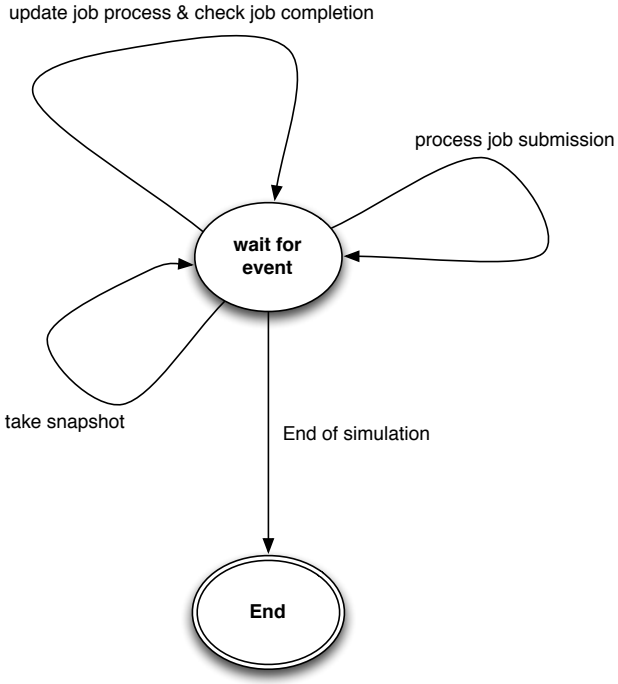

Figure 4.7: Utility allocation policy state machine

Update job processing is responsible for the progress of the job's execution and it is done every time an event is received. The first step is to calculate the time that passed between the last update time and the current instant of the simulation. The update only proceeds if the difference is greater than zero. Next, the resource's load is updated. The update is done using the same procedure used by the scheduler (see Equation 3.1). Once the update is done, the policy checks if there are jobs in execution. If there are no jobs executing the process ends. Otherwise, the jobs progress will be updated. The policy will iterate through the list of jobs in execution and calculate the work done by each job during the time interval that was calculated. The work W, in MI, done by a job during a time interval of T seconds, using the allocated PE MIPS capacity (M), is calculated using Equation 4.3. After the job's work is calculated, the policy decrements the remaining number of MI instructions in W MI's.

$$W = M \times T \tag{4.3}$$

Once all the jobs in execution are updated the process ends. Figure 4.8 shows the steps that were described earlier.

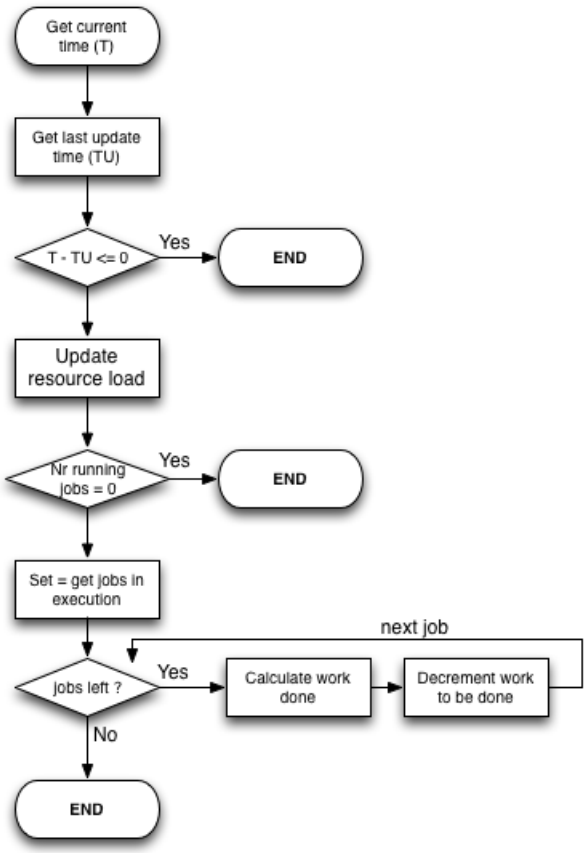

Figure 4.8: Update job processing

When a job submission request arrives to a resource, the policy must allocate the job to the machine and PE using the job's scheduling information data. The first step of this process is to update all the jobs in execution using the procedure depicted in Figure 4.8. After the update is finished, scheduling information such as selected machine and PE, required MIPS and utility value of each requirement, is retrieved from the job. Before allocating the job to the selected PE, the policy must check if that PE current state satisfies requisites present on scheduling information, i.e. verify if the PE's free MIPS is equal or bigger than the required MIPS that are present on the job's scheduling information.

A situation, in which the PE's free MIPS is smaller than the job's required MIPS, can happen if a job that is on the waiting queue is allocated to the PE that was selected for the incoming job before that job arrives to the resource. If the selected PE is unavailable, i.e. PE's free MIPS is smaller than job's required MIPS, the policy selects the PE that has the highest free MIPS. If all the PE are busy the job is put on the waiting queue and the job submission process ends. If a PE is available the policy recalculates the utility value of the finish time requirement and recalculates the global utility value. This procedure is important because the selected PE's free MIPS may cause the job's execution time value to change. If the selected PE is available or a substitute PE was found then the PE load is updated and the job is added to the list of jobs in execution. The final step is to calculate the next time jump. To do this, the completion time (CT) of all the jobs in execution is computed. Formula 4.4 shows how the completion time is calculated where CT is the completion time, L is the remaining job's MI that need to be processed and M is the MIPS that are being used by the job.

$$CT = \frac{L}{M} \tag{4.4}$$

If the smallest completion time of that set is greater than one second, it will be used as the next time jump value. Otherwise, the next time jump value is one second. This is done because some very small numbers may be rounded to zero and this would lead to an infinite loop. The steps that are taken when a job submission request arrives are depicted in figure 4.9.

Besides receiving job requests, the policy also handles internal events. Every time a resource receives an internal event, the policy takes two actions: update job processing and check job completion. The first action was already described and is depicted in Figure 4.8. The action related to checking if a job has completed and the all the inherent procedures will described next. First, the policy gets the list containing the jobs that are in execution. For each of those jobs, the policy checks if there is any work to be done. This implies verifying if the remaining MIs to be processed are equal to zero. If that is the case, the job is removed the execution list and the PE available MIPS is updated by removing the job from execution. After that, a snapshot of the resource is taken and the job results (utility value and execution time) are added to a message that is sent to the scheduler. Finally, if there are jobs on the waiting list the

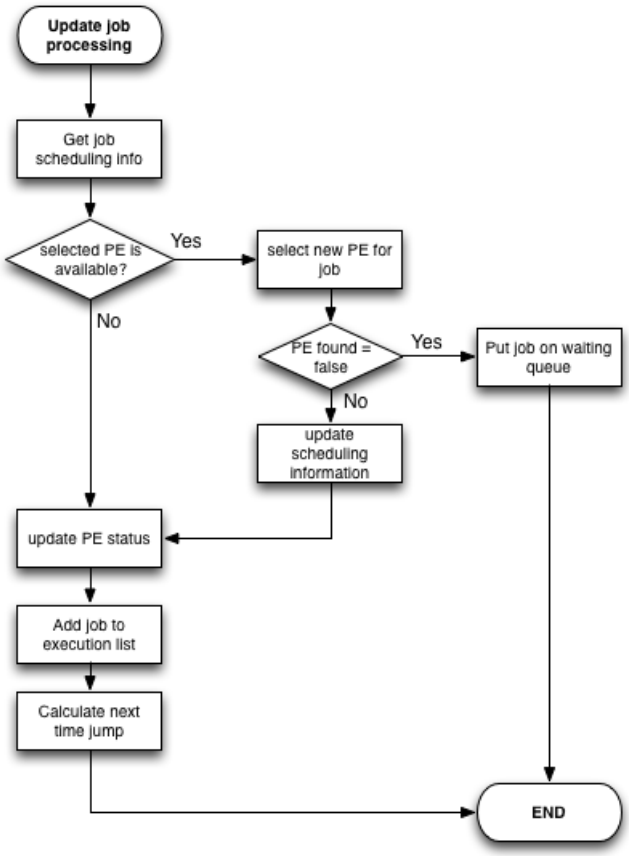

Figure 4.9: Job submission steps

policy tries to allocate then to a suitable PE. The described steps are depicted in Figure 4.10.

Periodic snapshots are internal events that are triggered every second (this value can be adjusted). The utility policy only starts taking periodic snapshots of the resources' states after the simulation setup time because during that time period no jobs are submitted and the resources are idle. To simplify the access, collection and statistical processing of data a class, called *ResourceSnapshot*, was created. Each time a periodic snapshot event is received, the policy will create a *ResourceSnapshot* object containing the following information: current simulation time, number of jobs in execution and their sizes and resource load. Once all this data is collected, the created *ResourceSnapshot* object is put into a list containing all the snapshots of that resource since the beginning of simulation. Finally, the policy schedules the next snapshot to the next second. Information from the snapshots will be used to analyze the performance of each algorithm in terms of resource utilization.

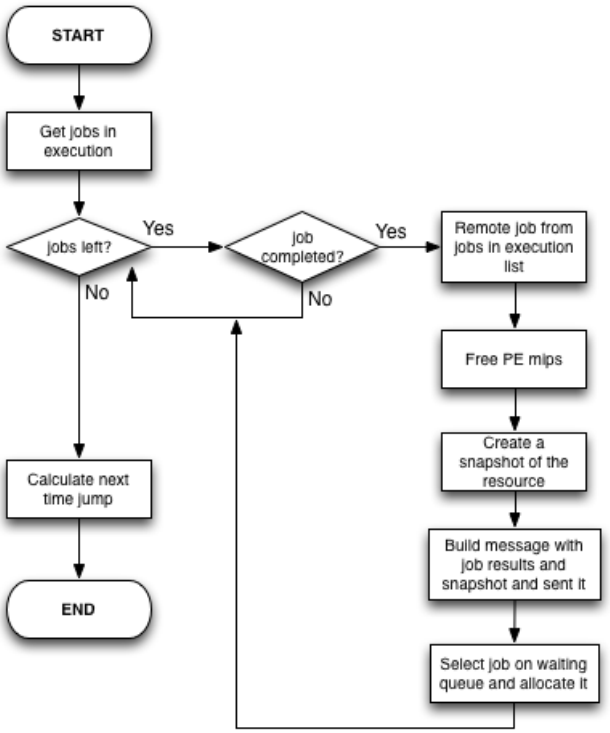

Figure 4.10: Check and process job completion

4.2.2 Other additions

Besides the *Scheduler* module other changes were made in the GridSim. This section will describe some new entities that were developed that are directly connected to the *Scheduler* module, as well as some changes that were made on existing entities due to the specifications of the solution. The rest of the section is organized as follows. Section 4.2.2.1 will describe *Message*, a new entity that was developed to simplify communication between entities. Section 4.2.2.2 will describe the new functionalities that were implemented on the *RegionalGIS* entity. Finally, section 4.2.2.3 will describe the new features of the *GridResource* entity.

4.2.2.1 Message

GridSim entities exchange information by passing serialized objects into their I/O ports. Using this feature, a special class, called *Message*, was created to encapsulates all the information exchanged between GridSim entities. Due to being used in all communications of all protocols that were developed, *Message* has a large set of attributes. Next, some of the most relevant attributes are presented.

- *Sender ID*: is an integer greater than zero that is the identification number of the entity that built the *Message*;

- *Stack of IDs*: a stack containing id's of GridSim entities. This stack is used when a message needs to be forwarded between different entities and those entities ids must be saved. An example is when a scheduler receives a request for a resource information from a remote GIS. On that situation, the stack will contain, from top to bottom: remote GIS ID, remote scheduler ID. The scheduler will pop the remote GIS ID and forward the message to it. The process will continue until the remote scheduler receives the information from the resource it requested;
- *GIS' IDs:* list containing IDs of the GIS in the grid. Used during simulation setup;
- *Job results*: object containing a set of important information about a job that has finished. This set includes the job's id, id of the resource were the job was executed, execution time, utility value and a boolean that indicates if the job is failed or not.

4.2.2.2 Regional GIS

The *RegionalGIS* (GIS) class was changed in order to support the behavior that the solution required. In the solution that is presented in this work, each *RegionalGIS* has a very important responsibility because it acts as a gateway of a VO, i.e every communication that is done between different VOs passes through the GIS. Next, there will a description of the additional modules that were added to the GIS class.

When selecting a resource to submit a job, the scheduler can select a local or remote resource. If the scheduler selects a remote resource, the job must be forwarded to the remote scheduler that belongs to the VO of the chosen resource. In this process, the GIS plays an important role. When a scheduler decides to forward a job to another VO it sends the message to its GIS, with a tag identifying it as a remote job submission message. The tag is called *REMOTE_JOB_SUBMISSION*. The message contains information about the job, the resource ID to where it should be submitted and the ID of the GIS of that resource. The GIS forwards the message to the remote resource's GIS. When the remote GIS receives the message, it identifies it as a remote job submission message and sends it to its local scheduler. When the scheduler receives the message it will start the remote job submission procedure. If the job is executed locally and finishes, the scheduler builds a message containing the job's results and a snapshot of the resource's state when the job ended. This message is sent to the local GIS with the intent of reaching the scheduler that was responsible for this remote submission. In order to achieve this, the message has a tag, *REMOTE_JOB_SUBMISSION_RESULTS*, and contains the ID of the GIS that is in the same VO as scheduler that was responsible for the remote submission. The local GIS uses the remote GIS id to forward the message. When the message is received by the remote GIS it is forwarded to its scheduler that uses the snapshot to update the view of the resource were the job was submitted and sends the results to the job's owner.

Resource refresh requests are crucial for each scheduler to have the best snapshot of a re-

mote resource's state. When a scheduler decides that a given entry of a resource needs to be refreshed, it creates a message with a specific tag and sends it to its local GIS. This tag is called *RESOURCE_REQUEST_REFRESH*. The message contains two important id's: the first is from the resource's and the second is from its GIS. Using the second id, the GIS forwards the message to the remote GIS. When the remote GIS receives the message, it identifies it as a resource refresh request. Next, it sends a message for that resource requesting information about its state. The resource creates a snapshot of its current state and sends it to its GIS. When the GIS receives the resource's snapshot it builds a message containing the snapshot and forwards it to GIS that sent the resource refresh request. This message is identified with a different tag, *RESOURCE_REFRESH_RESPONSE*. When the GIS receives the message, it identifies it as the response to a remote resource refresh request and forwards it to its local scheduler that extracts the resource's snapshot and updates the corresponding history entry.

The are two types of scheduling architectures in this work: centralized and decentralized. Some processes are executed in different ways in each architecture. To address this, some additional logic had to be added to some of the following GIS' processes: job submission and resource refresh. A description of these changes as well as the network architectures can be found in section 4.3

4.2.2.3 Grid Resource

The *GridResource* class was also modified. At a given time, a scheduler may need to request information from a resource in order to update its internal view, i.e. its history entry. This procedure implies, at a certain time, that the GIS requests a snapshot from the resource. To address such need, a new behavior was added to the *GridResource* class: refresh resource information. This new type of event is identified by the tag *REFRESH_RESOURCE_INFORMATION*. When a resource receives a refresh information event it needs to create a snapshot of its current state before sending it to the GIS. To do this, the resource creates a message that will contain the snapshot's information. The first step that is taken by the resource is to copy the resource's characteristics: id, architecture, operating system and machine information (machine id and list of PE). After copying the resource's characteristics, the next step is to calculate the resource load. This is done using the utility policy (see section 4.2.1.5 for more details). After that, the resource uses the utility policy to get information about the jobs that are in execution and the ones that are in the waiting queue. All this information is added to the message that was created and sent to the GIS.

Object copying and cloning

There are two ways of copying an object in Java: deep copy and shallow copy. Let's refer to A as the original instance and B as the cloned instance. In a deep copy, a new instance of the class is created (B) and all the fields of A are copied (using a new memory block) to that instance. This means that A and B do not share the same memory space, i.e. a change in A does not influences B and vice-versa. Figure

4.11 represents a deep copy example. In a shallow copy, a new instance is created (B) that shares the same memory space as the original (A). Figure 4.12 represents a shallow copy example.

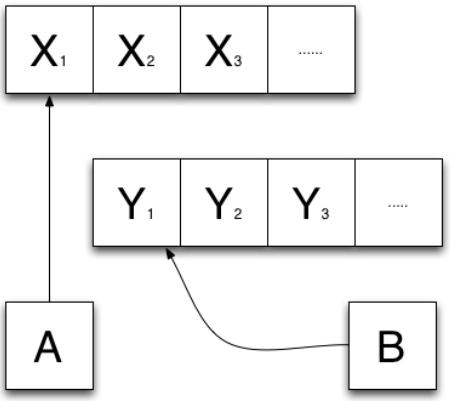

Figure 4.11: Deep copy

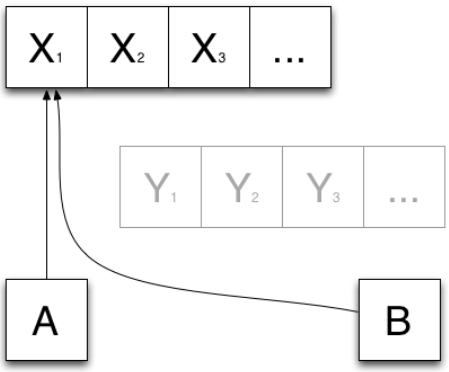

Figure 4.12: Shallow copy

Deep copy was the chosen approach to make the cloning of information that is required when building a snapshot. Although this approach requires more memory and, consequently, makes the simulation slower, it also represents a more realistic behavior. If the shallow copy was adopted, each time a resource's state changed all entities in the simulation would have information about that change instantaneously. The chosen solution (deep copy) is more consistent to what would happen in a real grid scenario.

4.3 Network Topology

GridSim has a module, called Net, that allows the creation of network topologies using GridSim entities. This module was used to create the network topology used in the simulations.

Each router was created using GridSim's RIPRouter class and was assigned a First In First Out (FIFO) packet scheduler using the simulator's FIFOScheduler class. This scheduler was chosen because all the packets are treated in the same way, i.e. there are no priorities. To connect routers to each other and to other entities the Link class was used. Each link's baud rate, MTU and propagation delay was chosen according to the connection type (intra or inter VO).

4.4 Synthesis and Final Remarks

This chapter described the implementation of our solution in the GridSim simulator. The original version of the simulator did not had a significant part of the functionalities required by our solution. Hence,

after presenting the simulator and its components, a detailed description of the modifications and additions to the simulator was done.

There was no support for a distributed-based scheduling architecture so it was necessary to create a class to address such need, the *Scheduler*, and to modify an exiting one, the *RegionalGIS*. The simulator did not provide any way to define job's requirements. A new class was created, *JobRequirements*, to overcome this limitation. GridSim defines two policies for allocating jobs to PEs. On the first one, there is no multitasking support, as each PE can only process one job at a time (space-shared). On the second one, there was multitasking support without priorities, as all jobs received the same *quantum*. Since jobs must have different priorities to met their execution time requirements, a new allocation policy was defined, *Utility Allocation Policy*. All these classes are not restricted to our solution and might be used in other scenarios.

Additionally, in order to support utility-based schedulers the class *Scheduling Information* was created to represent resource's utility information.

5 Evaluation

This chapter describes the simulations studies that were realized in order to address the quality of the proposed solution. It is organized as follows. Section 5.1 describes the simulation scenario. Section 5.2 presents the first set of simulations aimed to compare the performance of the different scheduling algorithm. A short validation of our decentralized architecture is performed in section 5.3. The final section 5.4 summarizes the results achieved and provides some final remarks.

5.1 Simulation scenario

5.1.1 Simulation goals

We have performed an intensive set of simulation studies focused in two main goals:

- *Goal I - Scheduling algorithms comparison* - Evaluation of the performance of PU scheduler algorithm by comparing it with other scheduling algorithms.
- *Goal II - Decentralized architecture validation* - Study of the key aspects of our architecture, namely the impact of the creation of VOs in the grid network.

5.1.2 Simulation setup

In order to have a flexible network architecture that allows us to simulate different scenarios, our simulator offers the possibility of customizing the grid network design by configuring the following set of input parameters:

- *Number of cluster* - defines the number of VOs.
- *Number of resources* - defines the number of resources assigned to each VO.
- *Number of users* - defines the number of users of the grid's network.
- *Number of jobs* - defines the number of jobs per user.

The list of resource attributes is predefined, comprising all the attributes that were considered at the design stage: *architecture*, *operating system*, *PE speed*, *number of PE* and *resource load*. The list of possible options is also predefined for each resource attribute, as well as the way they are assigned to each resource. For the sake of simplicity, we defined a limited set of options for each attribute. The information of the resource scenario is depicted in Table 5.1. As it can be stated in the table, our resource scenario is too restricted to represent the reality. Nonetheless, its use would not comprise the final results, because we are interested only in comparing the different algorithms under the same test conditions.

Resource attribute	Option	Selection function
Architecture	32 bits 64 bits	Uniform distribution
Operating system	MacOS Linux Solaris Windows	Uniform distribution
PE speed [MIPS]	[500..5000]	Uniform distribution
Number of PE	4	n.a.
Resource load [%]	[0.. 100]	f (simulation time)

Table 5.1: Resource's characterization

The *architecture*, *operating system* and *Maximum Execution Time* are the three job's requirements that we used. For the first two of them, the definition of the possible options uses the same approach that was used for the resources. Concerning the *Maximum Execution Time* , as jobs with different sizes will need different amount of time to be completed, its value depends on the job's size. Each one of the options has an utility value that states the user preference regarding that option. The assigned values ranges between [0,1] and it is not allowed that different options have the same utility value. Table 5.2. summarizes the relevant information.

Job requirements	Option	Selection function
Architecture	32 bits 64 bits	Uniform distribution
Operating system	MacOS Linux Solaris Windows	Uniform distribution
Maximum Execution time [s]	]0..Max]	Max = f(job's size)

Table 5.2: Job's requirements

To model job's submission process two other properties need to be defined: the *size* and the *inter-departure rate*. Using a similar approach of the one that was used by other author (Etminani & Naghibzadeh 2007; Chauhan & Joshi 2010), a simple model was used for the distribution of job's *size* and, for the *interdeparture rate*, a Poisson distribution function was used. The Poisson distribution as a parameter, *mean*, that represents the job's inter-departure rate, which is an input for the simulator. This

information is depicted in Table 5.3.

Job parameter	Option	Selection function
Size [MI]	[500..10000]	Uniform distribution
interdeparture rate [s]	Mean (Input parameter)	Poisson distribution

Table 5.3: Job's generation properties

For each experiment 10 simulation runs have been executed and statistically processed with a confidence level of 90%. The results that will be presented are the average values of these simulations. To guarantee the same conditions for each one the algorithms, the same seed is used in each one of the runs.

5.2 Goal I - Scheduling algorithms comparison

This section describes the first set of simulations, whose main goal is comparing the different scheduling algorithms and assess their relative performance. It will start by review the scheduling algorithms that are used. After, it presents the test methodology and performance metrics. Finally it provides a section for each group of experiments that haven been made to fulfill the goals that were defined.

5.2.1 Scheduling algorithms

The four different scheduling algorithms defined in chapter 3 have been simulated, namely:

1. PU - our proposal, that allows for partial fulfillment of user requirements.
2. BU - which provides binary scheduling decisions, as only one option per requirement is permited.
3. MM - that requires complete fulfillment of all user requirements.
4. RR - that does not take into account user requirements

5.2.2 Test methodology

Assessing the performance of the different scheduling algorithms is a very complex task due to the amount of information that might be configured, with impact in the performance. Hence, one of the first decisions that we took was the selection of the group of tests that enable us to understand if our proposal satisfies the requirements that have been defined. The following group of tests have been considered:

- *Group A* - Varying number of VOs .

- *Group B* - Varying number of jobs per user.
- *Group C* - Varying inter-departure rate.

In each one of the group, the parameter under analysis changes from simulation set to simulation set (simulation runs with different seeds), whist the other parameters are kept constant. Results are provided to each one of the individual tests, but the detailed analysis will be focused on a particular case. A comparative evaluation of the different tests is also provided. The entire set of results are included in Appendix 6.2.

5.2.3 Metrics

The following metrics will be used to validate the performance of the different scheduling algorithm.

- *Submit time*– average time since the user's jobs are created till they are submitted. The user's submission time is given by:

$$average\_time\_to\_submit\_job(u) = \frac{\sum_{j=1}^{n\_jobs_{(u)}} (t\_assign[j] - t\_submit[j])}{n\_jobs_{(u)}} \tag{5.1}$$

- *Execution time* – average time since the user's jobs are submitted till they are completely executed. It is given by:

$$average\_job\_execution\_time(u) = \frac{\sum_{j=1}^{n\_jobs_{(u)}} (t\_executed[j] - t\_assign[j])}{n\_jobs_{(u)}} \tag{5.2}$$

- *User utility* – average utility value of the resources used by the user's jobs. It is given by:

$$user\_util_{(u)} = 1 - \frac{\sqrt{\sum_{j=1}^{n\_jobs_{(u)}} utility\_value[j]}}{n\_jobs_{(u)}} \tag{5.3}$$

- User success ratio – is the ratio between the number of successfully completed jobs and the number of submitted jobs of each user. It is calculated as follows:

$$user\_suc\_ratio_{(u)} = \frac{\sum_{j=1}^{n\_jobs_{(u)}} complete[j]}{\sum_{j=1}^{n\_jobs_{(u)}} submit[j]} \tag{5.4}$$

In all the cases, the system values are given by averaging the values of all the users. The first two metric are used to study the network performance and the last two the user satisfaction.

5.2.4 Number of clusters (VO)

This section describes the results that were obtained when the number of VOs change. The main goal is to assess the impact of the grid organization on the network performance and user satisfaction. It start by the description of the simulation parameters, followed by the presentation and discussion of results.

5.2.4.1 Simulation parameters

Table 5.4 presents the simulation's parameters used in this group of tests.

Parameter	Value
Number of clusters	{2,4,8}
Number of resources	80
Number of users	70
Number of jobs	20
Inter-departure rate [s]	2

Table 5.4: Simulation variables: number of VOs

5.2.4.2 Individual test

Figure 5.1 depicts the results of the simulation tests that allows us to assess the network performance. The graphic of the left hand-side illustrates the average time to submit a job ($average\_time\_to\_submit\_job$) per user and the graphic of the right hand-side illustrates the average job execution time ($average\_job\_execution\_time$) per user. Each graphic represents the different scheduling algorithms under analysis.

A global analysis of the results shows there that are slightly variations of the values of the different users caused by the randomness of the job's submission process. The proposed solution, (PU), has an higher average time to submit a job than the others. This can be explained by the use of the forwarding mechanism between VOs, that lead to a remote job submission in case of having a remote resource that best fits user requirements. As none of the other algorithms use job forwarding, all of them have a lower average time to submit a job. Concerning the average job execution time, for the majority of users, our solution has the smallest value. Our solution provides the possibility of selecting the lowest loaded resource when more than one resource is able to satisfy the user requirements. Therefore, one can conclude that PU algorithm provides scheduling decisions that lead to a better network performance.

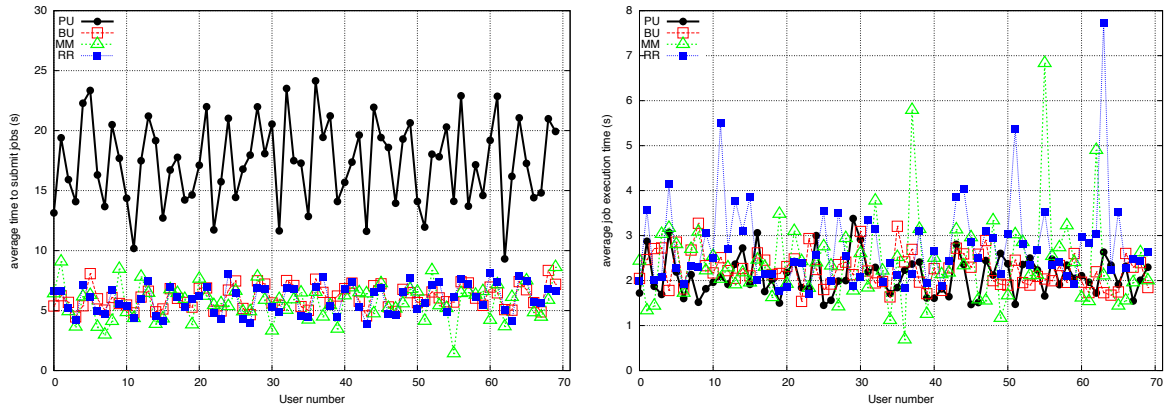

Figure 5.1: Two clusters: job submission and execution time per user

Figure 5.2 depicts the results of the simulation tests that allows to understand user satisfaction according to two different metrics: average user utility ($user\_util$), represented by the graph of the left hand-side, and the job's success ratio ($user\_suc\_ratio$), shown in the graph of the right hand-side.

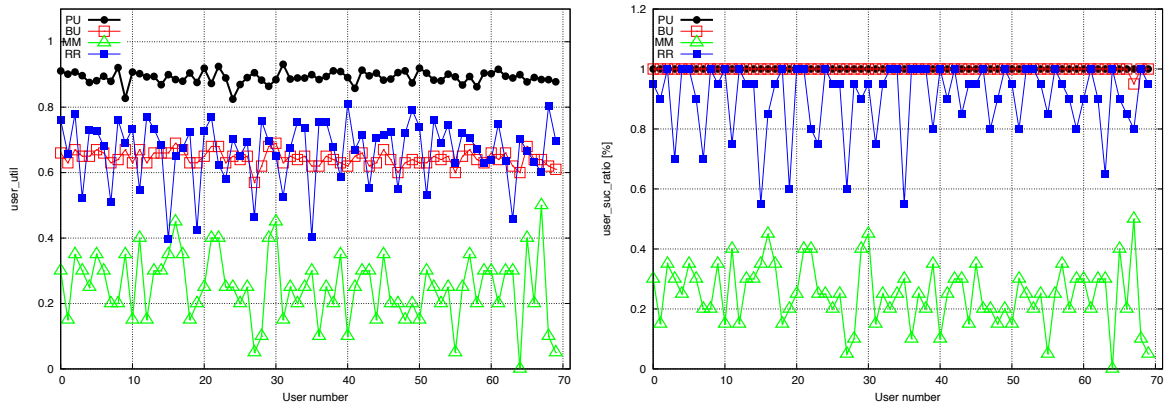

Figure 5.2: Two clusters: user's average utility and job success ratio

The analysis of the user utility shows that our PU scheduling algorithm (black serie) has always the highest utility, with no significant differences among the users and values near 0.9. The BU as similar behavior but with much lower utility values, near 0.6. Due to its characteristics, the RR has smaller values than PU and is the one that exhibits the biggest variation among the users. The MM scheduler presents the worst results, due to the difficulty of finding a resource available that matches exactly all the user's requirements. Concerning the success ratio, both PU and BU achieve very high values, whilst the other two scheduling algorithms (RR and MM) have a lower job success ratio.

A closer analysis of both graphics shows the utility value of job's success ratio of the MM are the same. In the MM algorithm, a job is submitted to a resource only if that resource satisfies all the job's requirements. If that happen, the job will be submitted and the utility value will be maximum. Otherwise, if no resource satisfies all the requirements, the job is declared as failed and its utility value is zero.

This explains why the utility values and job success ratio are the same.

5.2.4.3 Comparative results

The objective in this section is to study the influence of the variation of the number of VOs on each algorithm's performance. Other simulations were made with the four and eight VOs and the results are shown in Appendix 6.2, section A.1 . Here, we provide a comparative study of the system results achieved when the number of VOs. changes. Using a similar approach to the one provided in last section, we start by analyzing the network performance and after the user satisfaction.

Figure 5.3 represents the result of the studies used to assess the network performance variation with the number of VOs. The left hand-side depicts the system average time to submit a job ($System\_average\_time\_to\_submit$) and the right hand-side represents the system average execution time ($System\_average\_system\_execution\_time$).

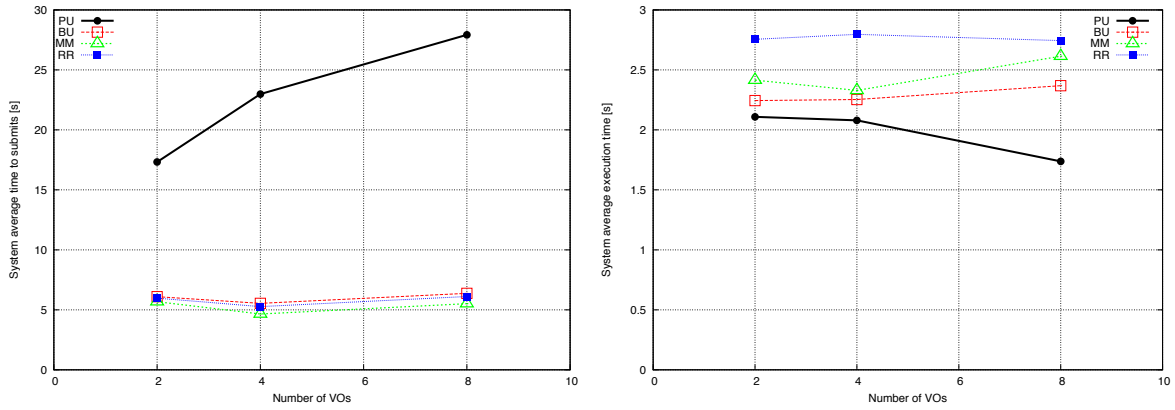

Figure 5.3: Cluster comparative analysis: average time to submit and average execution time

As expected, our proposal (PU), has an higher average time to submit a job, which increases with the number of VOs, due to job forwarding and no significant differences happened on the other scheduling algorithms. Concerning the average execution time, PU has the best results and, as the number of VOs increases the average execution time tends to decrease. The fact that the proposed solution has the job forwarding and partial utility mechanisms explain this behavior. If the local VO is very loaded, its local scheduler can forward a job to a least load (remote) resource and, therefore the execution time will be smaller.

Figure 5.4 depicts the variation of the user satisfaction when the number of VOs changes, with the left hand-side describing the utility value and the right hand-side the user's success ratio. The proposed solution (PU) has always the best utility value, which is kept constant with the number of VOs. Unlike PU, all the other scheduling algorithms provide much smaller utility, which decreases when the number of VOs increases. As PU is able to submit jobs to remote resources, the clustering of resources into more

VOs does not have any impact on it. However, this is not the case for the remaining algorithms, because resources outside the user's VO became unavailable. Concerning the success ratio, depicted in the graphic of the right hand-side, PU achieves almost 100% in all cases, and minor variations occur in the other algorithms.

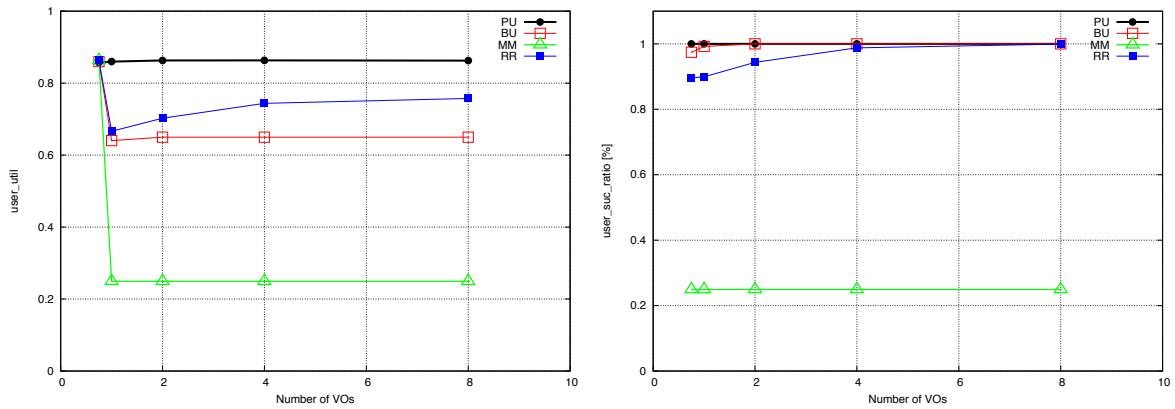

Figure 5.4: Cluster comparative analysis: average utility and job success ratio

5.2.5 Number of jobs

This section presents and describes the results that were obtained when the number of jobs per user changes. These tests measure the impact of the network load in the network performance and user satisfaction. The structures of the section is similar to the previous one.

5.2.5.1 Simulation parameters

Table 5.5 presents the simulation's parameters used in this group of tests.

Parameter	Value
Number of clusters	4
Number of resources	80
Number of users	70
Number of jobs	{5,10,20,30,40}
Inter-departure rate [s]	1

Table 5.5: Simulation variables: number of jobs

5.2.5.2 Individual test

Figures 5.5 depicts the average time to submit a job and the average execution time, when each user submits 20 jobs, at a mean rate of 1 job/s. Apart from RR that exhibits a more unstable behavior, mainly

because it does not use the knowledge of the network status in the process of job scheduling, all the others algorithms present the same type of behavior that was described before.

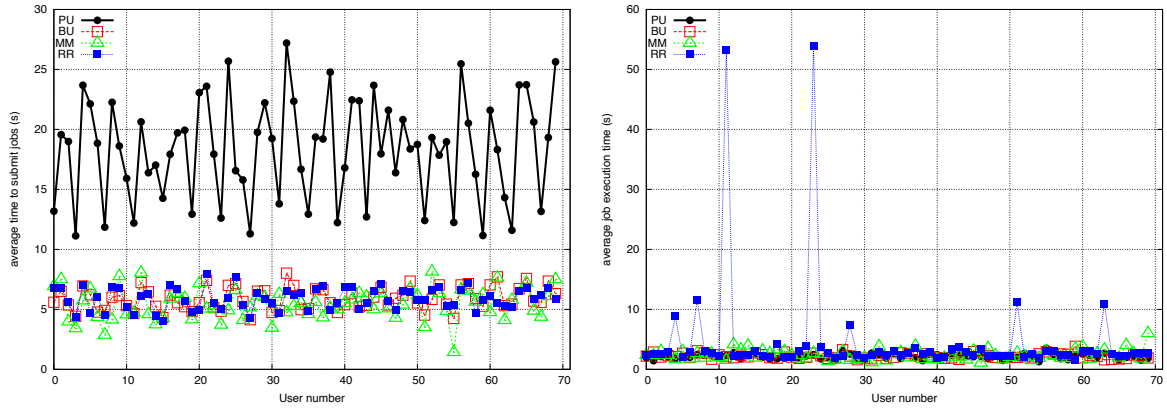

Figure 5.5: Jobs per user 20: average time to submit and average execution time

Figure 5.6 illustrates the utility value and the success ratio of this test. An analysis of the average utility shows, once again, that the proposed solution, (PU), has an higher utility than the other algorithms. Concerning the success ratio, it is clear that PU has the highest levels of success. The MM has the worst performance and RR maintains its inconstant behavior. Another important detail is the fact that, for some number of users, the MM algorithm shows some success ratio that are near zero.

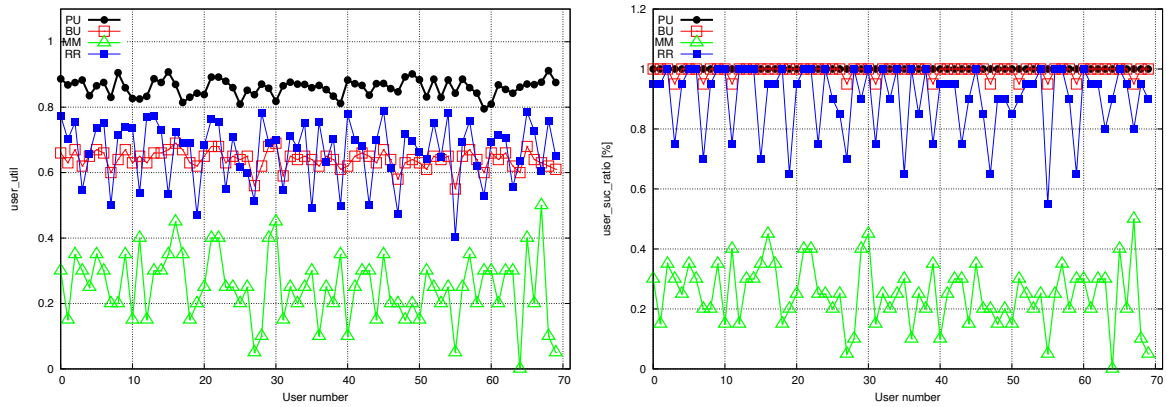

Figure 5.6: Jobs per user 20: user's average utility and job success ratio

5.2.5.3 Comparative analysis

Five different number of jobs per user values were used in various simulations: 5, 10, 20, 30 and 40 jobs per user. The results of the individual tests are represented in Appendix 6.2, section A.2. The objective in this section is to study the influence of the variation of the number of jobs per user on each algorithm's performance. Using the same approach that was used before, we start by analyzing the network performance and after the user satisfaction.

Figures 5.7 and 5.8 present the results. On the left hand-side of figure 5.7 there is a graphic that shows the evolution of the average job submission time when the number of jobs changes. As the number of jobs per user increase so does the average time to submit. In PU scheduling algorithm, the average time to submit a job is higher due to the forwarding mechanism, as explained before. The graphic on the right hand-side of the same figure shows that our proposal (PU) has always the smallest average job execution time.

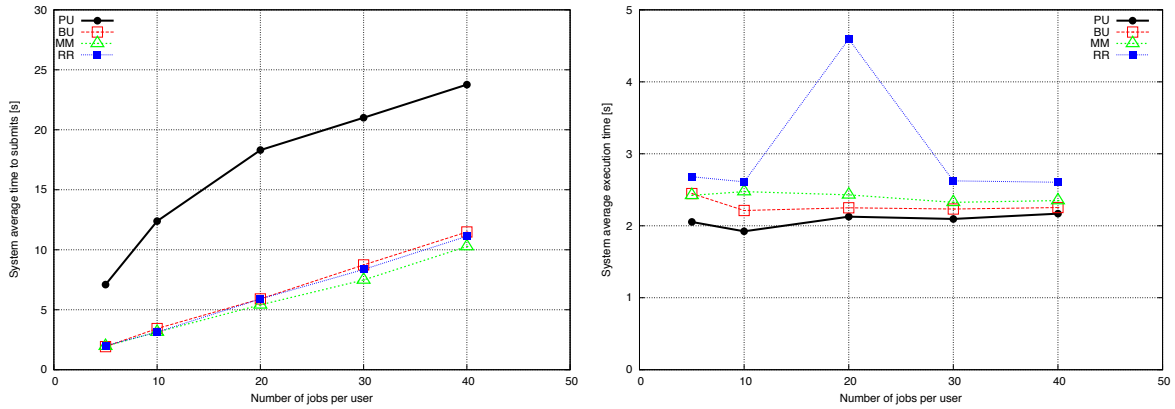

Figure 5.7: Number of jobs per user comparison: average time to submit and average execution time

Figure 5.8 depicts two graphics that represent the average utility (on the left hand-side) and the average success ratio (on the right hand-side). An analysis of the average utility shows, once again, that the proposed solution has an higher utility than the others, even with a decrease of performance when the number of jobs per user increases. This means that the network is becoming more loaded and it would not be possible to submit all jobs to resources that match the users' best preferences. However, even in a more loaded network, PU algorithm is the only one that is able to satisfy almost 100% of the jobs. All the remaining algorithms experience much smaller values, even the BU that, in the previous studies had a very good performance, is decreasing its success ratio.

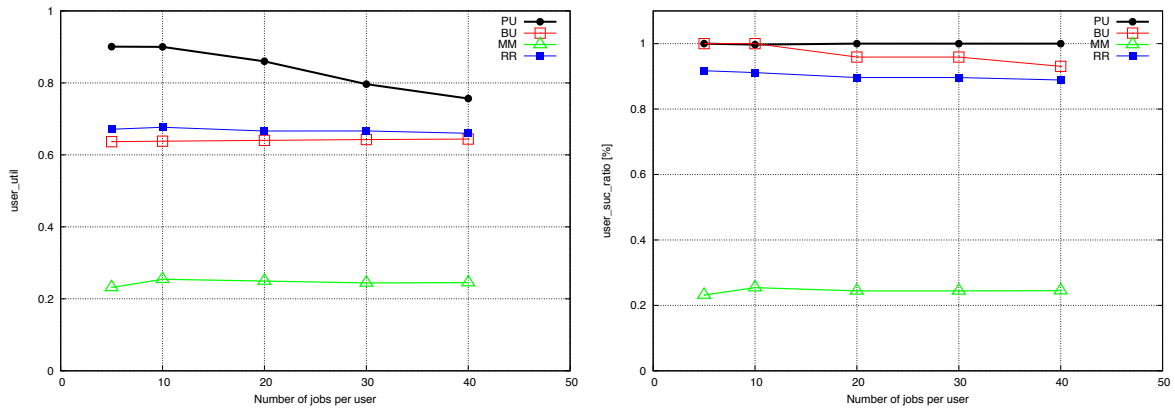

Figure 5.8: Number of jobs per user comparison: user's average utility and job success ratio

5.2.6 Job Inter-departure rate

This section presents and describes the results of changing the job inter-departure time, that corresponds to the mean value of the the poisson distribution. The goal is to increase the network load by creating the conditions to increase concurrency. The structures of the section is similar to the previous one.

5.2.6.1 Simulation parameters

Table 5.6 presents the simulation's parameters used in this group of tests.

Parameter	Value
Number of cluster	4
Number of resources	80
Number of users	70
Number of jobs	20
Inter-departure rate [s]	{0.75,1,2,4,8}

Table 5.6: Simulation variables: inter-departure time

5.2.6.2 Individual analysis

Figures 5.9 present the results that were obtained with an inter-departure rate of 0.75 s for the average time to submit and average execution time. No significant differences were found when compared with the results of other previous tests, showing different behavior for different users and the same trend among the algorithms. Once again, PU has the highest submission time and a sightly better execution time for some users.

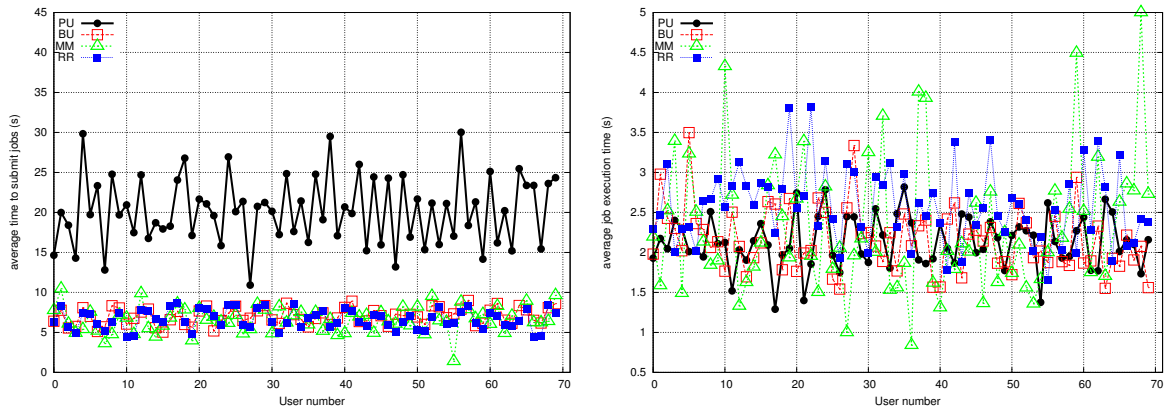

Figure 5.9: Inter-departure rate 0.75 s: average time to submit and average execution time

Regarding the average utility value and the average success ratio, shown in Figure 5.10, the situation is similar, with the algorithms ranked in the same way.

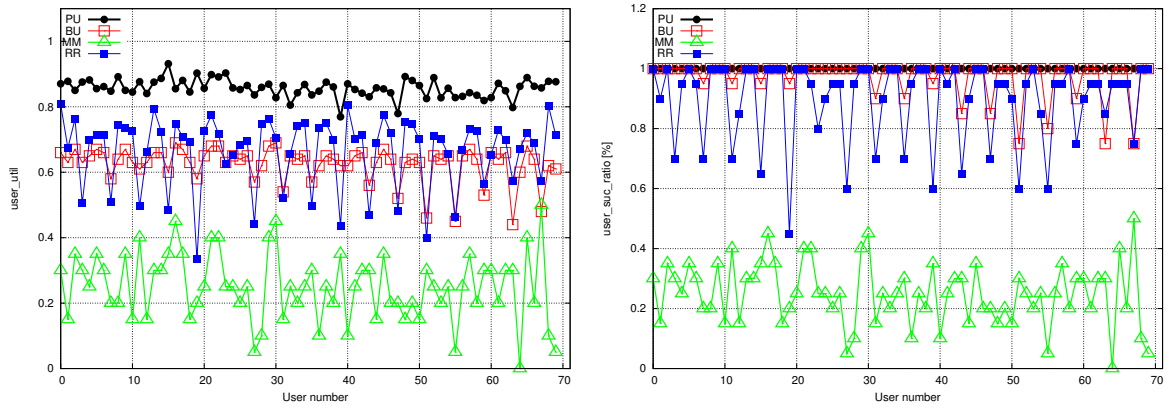

Figure 5.10: Inter-departure rate 0.75 s user's average utility and job success ratio

5.2.6.3 Comparative analysis

Although all the results of the individual tests are represented in Appendix 6.2, section A.3, in this section a comparative analysis is performed.

Figure 5.11 depicts the evolution of the system average time to submit (left hand-side) and average execution time (right hand-side) with the inter-departure rate. The analysis of the evolution of the system average time to submit shows that with lower inter-departure time, such as 0.75, 1 and 2, the PU scheduling algorithm has an higher average time to submit values than the other solutions. However, as the rate increases the algorithm tends to exhibit a behavior similar to the others. This happens because on lower levels of the inter-departure time, jobs arrive at a quicker rate, local resources become loaded and the proposed solution will forward to jobs to less loaded VOs. Concerning the execution time, in all the cases, our scheduler offers the smallest time, although the difference is not very big.

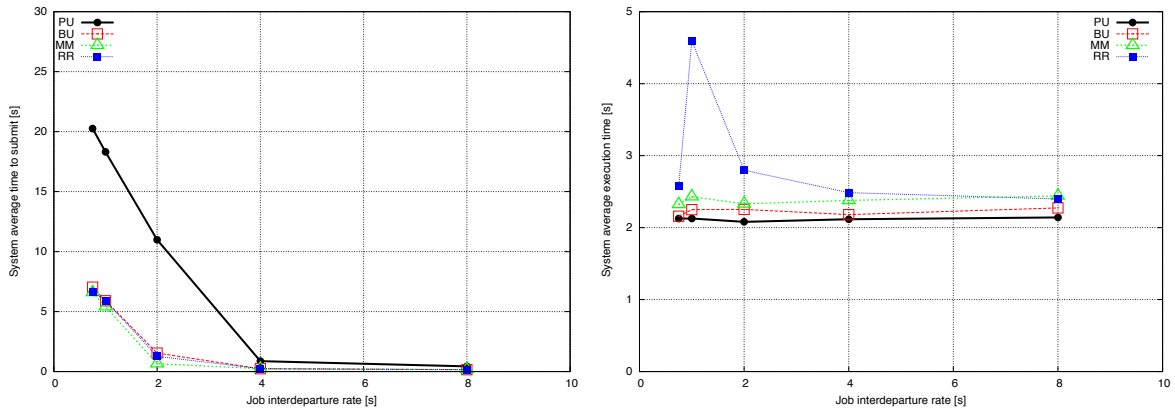

Figure 5.11: Poisson comparative analysis: average time to submit and average execution time

The user satisfaction is illustrated in figure 5.12 . It represents the variation of the utility value and success ratio with the inter-departure rate. Once again, our PU algorithm outperforms the others,

maintaining the highest values of utility and success ratio, even when the rate is small. It is possible to verify that other solutions, do not perform so well. When inter-departure rate is small, the success ratio of other algorithms, namely the BU is smaller than when rate the increases.

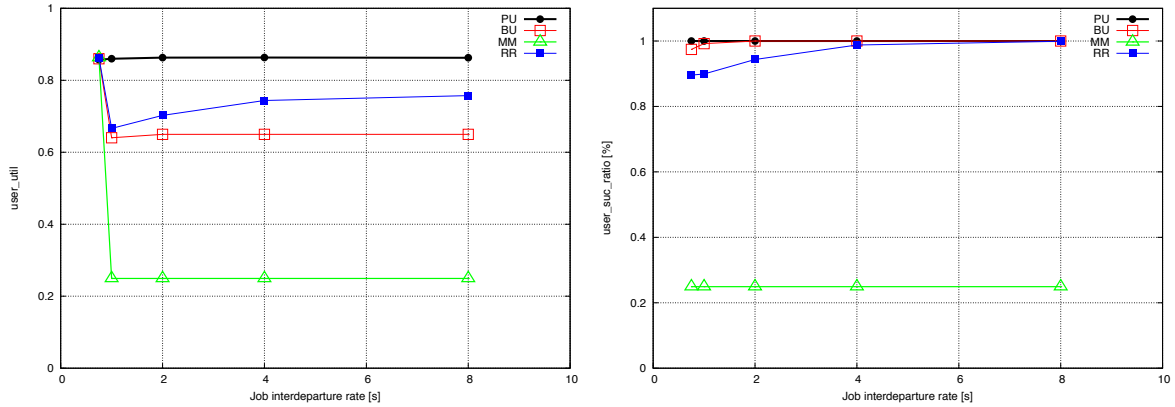

Figure 5.12: Poisson comparative analysis: user's average utility and job success ratio

5.3 Goal II - Decentralized architecture validation

This section describes the second set of simulations, which is main goal is a preliminary validation of the decentralized architecture. It start by presenting the test methodology and performance metrics, and after describes the tests and discusses the results.

5.3.1 Test methodology

Several important aspects may be assessed to evaluate the performance of our decentralized architecture. One of them, the impact of the number of VOs, has already been studied in the previous section. So, in this section we studied the VO performance, in terms of network load and load balancing. No comparison with other schedulers is provided, as we want to focus our attention only in our proposal.

5.3.2 Performance metrics

The following metrics will be used to validate the performance of our scheduling algorithm.

- VO load - The average load of the VO during a given interval of time. It is measured as follows:

$$VO\_load(v,t) = \frac{\sum_{j=1}^{n\_res_{(v)}} load_{(j,t)}}{n\_res_{(v)}} \tag{5.5}$$

- VO load balancing level– The load variation of the VO during a given interval of time. It is measured as follows:

$$VO\_load\_balance(v) = 1 - \frac{std\_dev\_load(v)}{avg\_load(v)} \tag{5.6}$$

Where avg_load represents the mean resource utilization and std_dev_load the respective mean square deviation.

5.3.3 Load distribution

5.3.3.1 Simulation parameters

Table 5.7 presents the simulation's parameters used in this group of tests.

Parameter	Value
Number of clusters	4
Number of resources	80
Number of users	70
Number of jobs	20
Inter-departure rate [s]	1

Table 5.7: Simulation variables: inter-departure time

5.3.3.2 Comparative analysis

Figure 5.13 describes the load of a VO during time ($VO\_load(v,t)$), and the associated load balance, ($VO\_load\_balance(v)$). The first one is depicted in the right hand-side and, the second one, in the left.

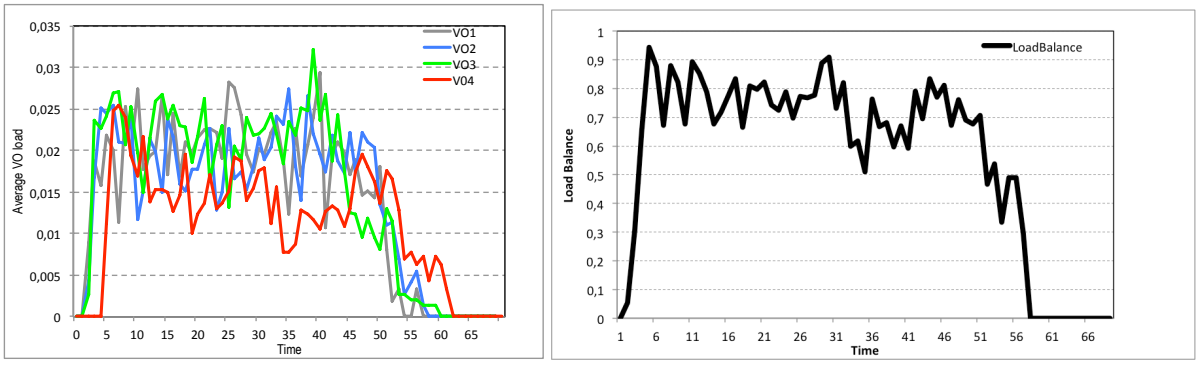

Figure 5.13: Load distribution per VO and Load Balancing

The analysis of the load distribution per VO shows that VOs have a very low average resource load, which changes during the simulation time. At the beginning of the simulation, network resources are

free and quickly achieves a traffic peak that last until time 50s, due to job's submission. At this instant, the load starts to decrease, because most of the jobs finish their execution and few new jobs arrive. The different loads that several VOs have are caused by the execution of long jobs.

When comparing the load of different VOs it is possible to conclude that they exhibit mostly the same type of trend, without significant differences among them. Hence, network is quite well balanced, as the graphic on the left hand-side shows.

5.4 Synthesis and final remarks

This chapter describes the simulation studies that have been made to assess the performance of our solution.

In the first part, the different algorithms (PU, BU, MM and RR) have been tested using the same conditions. The tests includes variations of different parameters, keeping the others constant. The number of VOs, the number of jobs per users and the job inter-departure rate have been tested. In all the cases the average time to submit a job and the average execution time was used to asses the network performance and the average utility and success ratio to assess the user satisfaction.

In terms of performance, our proposal (PU) presents an higher time to submit a job than the others, due to the possibility of performing job forwarding and remote submissions. This value increases with the number of VOs and network load (higher number of jobs per user or smaller inter-departure time). PU exhibits the smaller average execution time, which significantly decreases with the number of VOs and remains almost constant with the network load.

In terms of user satisfaction, our proposal outperforms the others in both metrics: the utility value is always above 0.85, whilst BU and RR values stays around 0.6 The worst results are achieved by MM (above 0.2) due to the restricted scheduling policy. Job success ratio of PU is near 100% and the other has some variations. The policy decision used by RR leads to some unpredictable behavior, which might be motivated by the absence of an adequate resource allocation strategy.

In the second part, a preliminary assessment of the decentralized topology was made. The tests focused exclusively on PU scheduler and measures load distribution per VO and load balancing.

The results showed that, with a very low network load, it is possible to achieve a good load balance. However, there are variations between the VOs, that negatively impacts this metric and might be caused by the execution of long jobs.

Limitations of the core of the simulator does not allow us to run tests with more loaded networks. However, if such is the case, the performance of PU would be even better (when compared with the others) and the load more balanced.

6 Conclusions and future work

6.1 Conclusions

This thesis proposes a decentralized scheduling architecture that aims at improving network performance an users' satisfaction. To do so, we proposed a decentralized scheduling architecture were each VO maintains the information of both local and remote resources through the use of a GIS. Our architecture also comprises a scheduler that is responsible for all scheduling mechanisms. It contains a *Resource Manager*, to monitor the network resources, and a *Job Scheduler*, to perform the scheduling decisions and select wether a job must be locally or remotely submitted. Users specify their requirements during job creation and rank the different options according to their preferences (*Partial Utility*. This information will be used to assist the scheduling decisions, together with that provided by the resource usage.

The architecture was implemented in GridSim, which is a complex simulator that requires a significant amount of time to understand so that modifications can be implemented. Nevertheless, an extensive set of features were added through the creation of new classes or modifications of existing ones, not only to support our model, but also to provide a more realistic simulation environment. So, to support the distributed scheduling architecture a few new classes were added, such as *Scheduler* and *JobRequirement*. GridSim defines two policies for allocating jobs to PEs, but does not support multi-task with priorities. Since jobs must have different priorities to met their execution time requirements, a new allocation policy was defined, *Utility Allocation Policy*.

Simulation studies comprises the comparison of our proposal with other scheduling algorithms and a preliminary assessment of the decentralized architecture. The performance results shows that our scheduler, PU, outperforms the others in terms of average execution time, utility and success ratio. Nevertheless, the possibility of remote job forwarding lead to an higher average time to submit a job.

The simulation studies shows also that, with a very low network load, it is possible to achieve a good load balance. Limitations of the core of the simulator does not allow us to run tests with more loaded networks.

6.2 Future work

After performing this work, there is still space for improvement and additional validations.

From an architectural perspective there are a few open issues. One of the most simple to achieve is the prioritization of user requirements. The management of the grid network was not considered in our solution. Hence, failure or discovery of new resources or schedulers is not handle and limited reliability and redundancy can be offered. These aspects should be addressed in the future.

From a simulation perspective, it will be interesting to evaluate the results in more complex networks. Another interesting issue, is modeling a real grid scenario, with the existing resources and real job traces.

Also as future work, the implementation of this solution in a real grid system is an interesting topic of research.

Bibliography

Abramson, D., R. Buyya, & J. Giddy (2002). A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems 18*, 1061–1074.

Amudha, T. & T. Dhivyaprabha (2011, June). Qos priority based scheduling algorithm and proposed framework for task scheduling in a grid environment. In *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, pp. 650 –655.

Bester, J., I. Foster, C. Kesselman, J. Tedesco, & S. Tuecke (1999). Gass: a data movement and access service for wide area computing systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, IOPADS '99, New York, NY, USA, pp. 78–88. ACM.

Buyya, R., D. Abramson, & J. Giddy (2000). Nimrod/G: an Architecture for a Resource Management and Scheduling System in a Global Computational Grid. *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 283–289 vol.1.

Buyya, R., D. Abramson, J. Giddy, & H. Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing 2 . Players in the Grid Marketplace. *Marketplace*, 1–27.

Buyya, R. & M. Murshed (2002, November). GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience 14*(13-15), 1175–1220.

Casavant, T. & J. Kuhl (1988, feb). A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *Software Engineering, IEEE Transactions on 14*(2), 141 –154.

Chauhan, S. & R. Joshi (2010, feb.). A weighted mean time min-min max-min selective scheduling strategy for independent tasks on grid. In *Advance Computing Conference (IACC), 2010 IEEE 2nd International*, pp. 4 –9.

Chen, D., G. Chang, X. Zheng, D. Sun, J. Li, & X. Wang (2011). A novel p2p based grid resource discovery model. *Journal of Networks 6*(10).

Chen, J. (2010, April). Economic grid resource scheduling based on utility optimization. In *Intelligent Information Technology and Security Informatics (IITSI), 2010 Third International Symposium on*, pp. 522 –525.

Chervenak, A., E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, & B. Tierney (2002, Nov.). Giggle: A framework for constructing scalable replica location services. In *Supercomputing, ACM/IEEE 2002 Conference*, pp. 58.

Chunlin, L. & L. Layuan (2005, March). A distributed utility-based two level market solution for optimal resource scheduling in computational grid. *Parallel Computing 31*(3-4), 332–351.

Chunlin, L. & L. Layuan (2007). An optimization approach for decentralized qos-based scheduling based on utility and pricing in grid computing. *Concurrency Computation Practice And Experience 19*(1), 107–128.

Cooper, K., A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, & C. Mendes (2005). New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming 33*, 209–229.

Czajkowski, K., S. Fitzgerald, I. Foster, & C. Kesselman (2001). Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pp. 181 –194.

Czajkowski, K., I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, & S. Tuecke (1998). A resource management architecture for metacomputing systems. In D. Feitelson & L. Rudolph (Eds.), *Job Scheduling Strategies for Parallel Processing*, Volume 1459 of *Lecture Notes in Computer Science*, pp. 62–82. Springer Berlin - Heidelberg.

Deelman, E., J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, & M. Livny (2004). Pegasus: Mapping scientific workflows onto the grid. In M. Dikaiakos (Ed.), *Grid Computing*, Volume 3165 of *Lecture Notes in Computer Science*, pp. 131–140. Springer Berlin - Heidelberg.

Deelman, E., G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, & D. S. Katz (2005, July). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program. 13*, 219–237.

Dong, F. & S. G. Akl (2006). Scheduling Algorithms for Grid Computing : State of the Art and Open Problems. *Components*, 1–55.

Etminani, K. & M. Naghibzadeh (2007, sept.). A min-min max-min selective algorihtm for grid task scheduling. In *Internet, 2007. ICI 2007. 3rd IEEE/IFIP International Conference in Central Asia on*, pp. 1 –7.

Fahringer, T., A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, & H.-L. Truong (2005). Askalon: a tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience 17*(2-4), 143–169.

Fahringer, T., R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, & M. Wieczorek (2007). Askalon: A development and grid computing environment for scientific workflows. In I. J. Taylor, E. Deelman, D. B. Gannon, & M. Shields (Eds.), *Workflows for e-Science*, pp. 450–471. Springer London.

Fahringer, T., J. Qin, & S. Hainzer (2005, may). Specification of grid workflow applications with agwl: an abstract grid workflow language. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, Volume 2, pp. 676 – 685 Vol. 2.

Fahringer, T. & C. Seragiotto (2002). Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In S. Sahni, V. Prasanna, & U. Shukla (Eds.), *High Performance Computing - HiPC 2002*, Lecture Notes in Computer Science, pp. 151–162. Springer Berlin - Heidelberg.

Foster, I. & C. Kesselman (1996). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications 11*, 115–128.

Foster, I., C. Kesselman, G. Tsudik, & S. Tuecke (1998). A security architecture for computational grids. In *Proceedings of the 5th ACM conference on Computer and communications security*, CCS '98, New York, NY, USA, pp. 83–92. ACM.

Foster, I., J. Vockler, M. Wilde, & Y. Zhao (2002). Chimera: a virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pp. 37 – 46.

Frey, J., T. Tannenbaum, M. Livny, I. Foster, & S. Tuecke (2002). Condor-G: a computation management agent for multi-institutional grids. *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, 55–63.

Gomes Ramos, T. & A. Magalhaes Alves de Melo (2006, May). An extensible resource discovery mechanism for grid computing environments. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, Volume 1, pp. 115 – 122.

Howell, F. & R. Mcnab (1998). simjava: A discrete event simulation library for java. In *In International Conference on Web-Based Modeling and Simulation*, pp. 51–56.

Huedo, E., R. Montero, & I. Llorente (2004, Feb.). Experiences on adaptive grid scheduling of parameter sweep applications. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pp. 28 – 33.

Imamagic, E., B. Radic, & D. Dobrenic (2006a). An Approach to Grid Scheduling by Using Condor-G Matchmaking. *Journal of Computing and Information Technology*, 329–336.

Imamagic, E., B. Radic, & D. Dobrenic (2006b). An Approach to Grid Scheduling by using Condor-G Matchmaking Mechanism. *28th International Conference on Information Technology Interfaces, 2006.* (3), 625–632.

Izakian, H., A. Abraham, & V. Snasel (2009, april). Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, Volume 1, pp. 8 –12.

Jacob, J., E. B. Rajsingh, & I. B. Jesudasan (2011). Dynamic multi dimensional matchmaking model for resource allocation in grid environment. In D. Nagamalai, E. Renault, & M. Dhanuskodi (Eds.), *Trends in Computer Science, Engineering and Information Technology*, Volume 204 of *Communications in Computer and Information Science*, pp. 179–185. Springer Berlin Heidelberg.

Kaur, E. & J. Sengupta. Resource discovery in web-services based grids.

Krauter, K., R. Buyya, & M. Maheswaran (2002, February). A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience 32*(2), 135–164.

Kurowski, K., J. Nabrzyski, A. Oleksiak, & J. Weglarz (2007, dec.). Grid scheduling simulations with gssim. In *Parallel and Distributed Systems, 2007 International Conference on*, Volume 2, pp. 1 –8.

Lee, Y.-H., S. Leu, & R.-S. Chang (2011, October). Improving job scheduling algorithms in a grid environment. *Future Generation Computer Systems 27*(8), 991–998.

Legrand, A., L. Marchal, & H. Casanova (2003, may). Scheduling distributed applications: the simgrid simulation framework. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pp. 138 – 145.

Ma, S., X. Sun, & Z. Guo (2010, July). A resource discovery mechanism integrating p2p and grid. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, Volume 7, pp. 336 –339.

Maheswaran, M., S. Ali, H. Siegal, D. Hensgen, & R. Freund (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pp. 30 –44.

Othman, A., P. Dew, K. Djemame, & I. Gourlay (2003, sept.). Adaptive grid resource brokering. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 172 –179.

Prodan, R. & T. Fahringer (2004). Zenturio: a grid middleware-based tool for experiment management of parallel and distributed applications. *Journal of Parallel and Distributed Computing 64*(6), 693 – 707.

Rahman, M., R. Ranjan, R. Buyya, & B. Benatallah (2011). A taxonomy and survey on autonomic management of applications in grid computing environments. *Library* (May), 1990–2019.

Raman, R., M. Livny, & M. Solomon (2000). Resource management through multilateral matchmaking. *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, 290–291.

Sahu, R. (2011). Many-Objective Comparison of Twelve Grid Scheduling Heuristics. *International Journal 13*(6), 9–17.

Schopf, J. M. (2004). *Chapter 1 Ten Actions when Grid Scheduling The User as a Grid Scheduler*.

Silva, J., P. Ferreira, & L. Veiga (2010, april). Service and resource discovery in cycle-sharing environments with a utility algebra. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –11.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001, August). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev. 31*, 149–160.

Takefusa, A., S. Matsuoka, H. Nakada, K. Aida, & U. Nagashima (1999). Overview of a performance evaluation system for global computing scheduling algorithms. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pp. 97 –104.

Tanenbaum, A. S. (2007). *Modern Operating Systems* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall Press.

Thain, D., T. Tannenbaum, & M. Livny (2003). *Condor and the Grid*, pp. 299–335. John Wiley and Sons, Ltd.

Truong, H.-L. & T. Fahringer (2002). Scalea: A performance analysis tool for distributed and parallel programs. In B. Monien & R. Feldmann (Eds.), *Euro-Par 2002 Parallel Processing*, Volume 2400 of *Lecture Notes in Computer Science*, pp. 41–55. Springer Berlin / Heidelberg.

Vasques, J. a. & L. Veiga (2013). A decentralized utility-based grid scheduling algorithm. In *SAC*. ACM.

Xhafa, F. & A. Abraham (2008). Meta-heuristics for Grid Scheduling Problems. pp. 1–37.

Xhafa, F. & A. Abraham (2010, April). Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems 26*(4), 608–621.

Zhang, X., J. Freschl, & J. Schopf (2003, June). A performance study of monitoring and information services for distributed systems. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pp. 270 – 281.

Zhao, Y., M. Wilde, I. Foster, J. Voeckler, J. Dobson, E. Gilbert, T. Jordan, & E. Quigg (2006). Virtual data grid middleware services for data-intensive science. *Concurrency and Computation: Practice and Experience 18*(6), 595–608.

A Simulation results

A.1 Number of clusters

A.1.1 Metrics - job submission and execution time

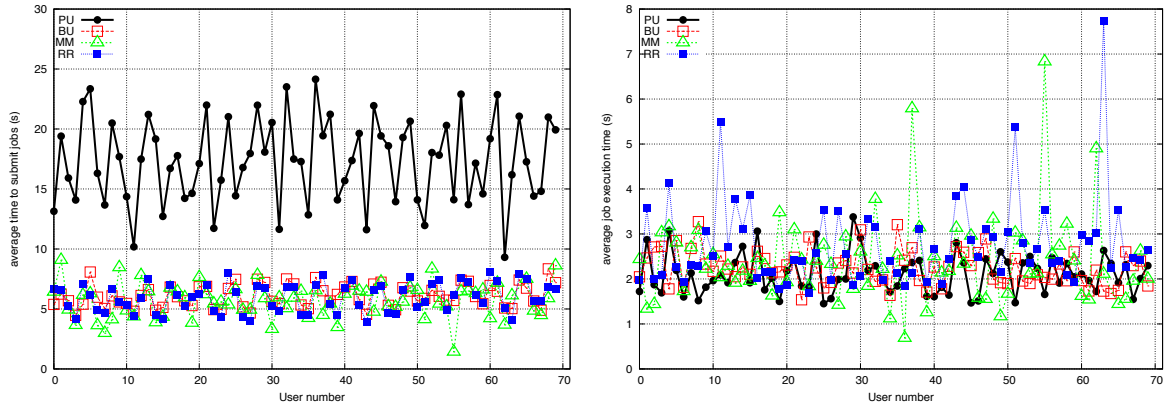

Figure A.1: Two clusters: job submission and execution time per user

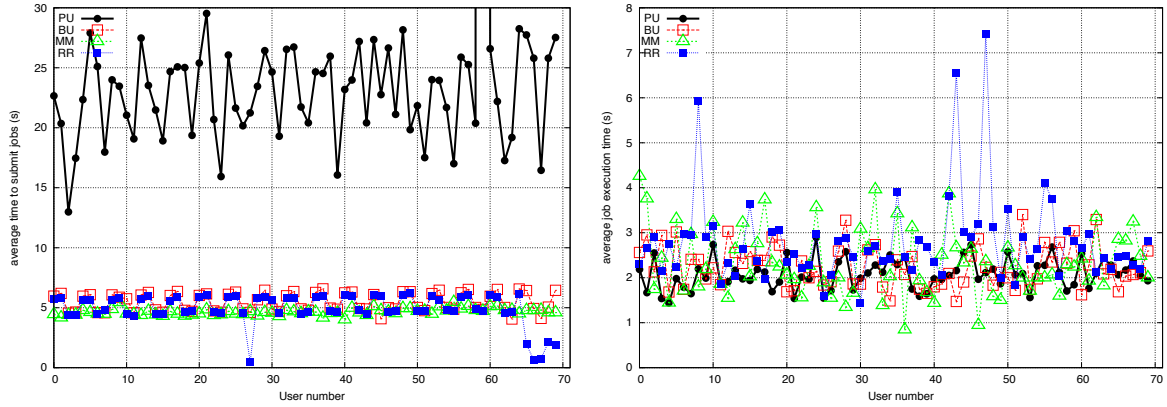

Figure A.2: Four clusters: job submission and execution time per user

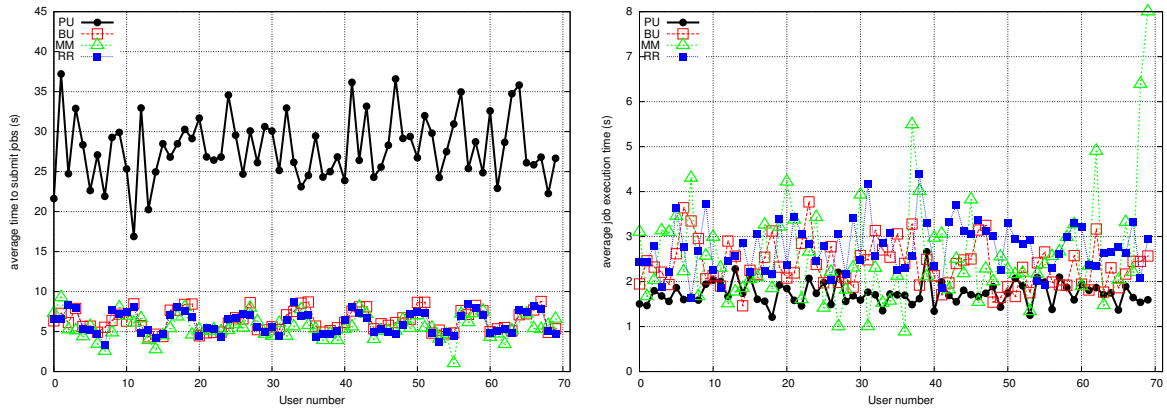

Figure A.3: Eight clusters: job submission and execution time per user

A.1.2 Metrics -user's average utility and job success ratio

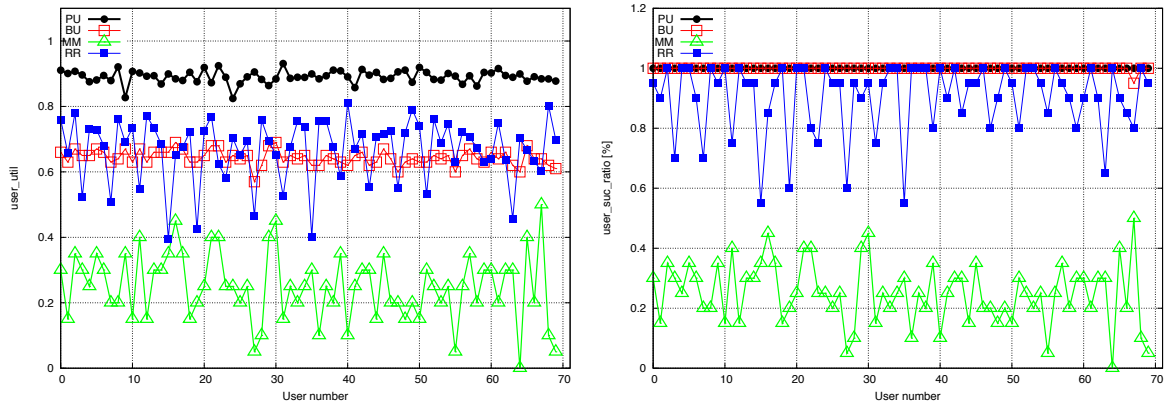

Figure A.4: Two clusters: user's average utility and job success ratio

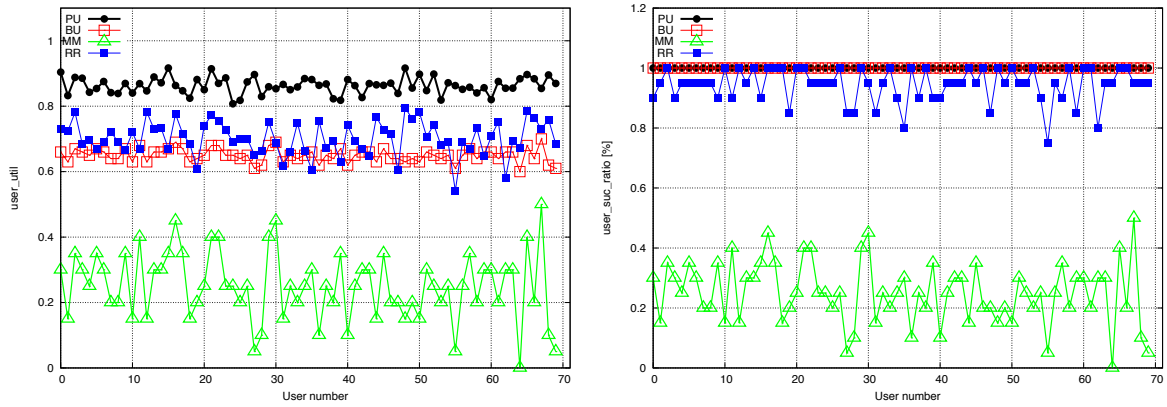

Figure A.5: Four clusters: user's average utility and job success ratio

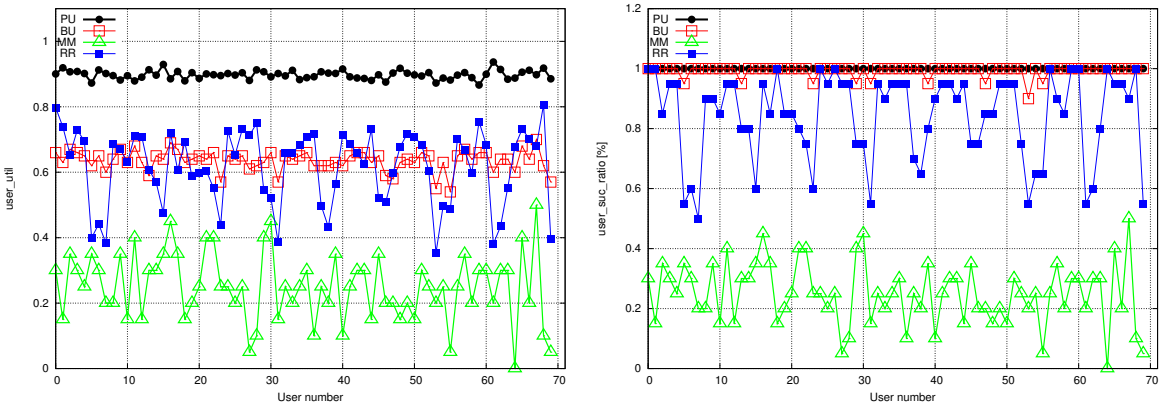

Figure A.6: Eight clusters: user's average utility and job success ratio

A.2 Number of jobs

A.2.1 Metrics - job submission and execution time

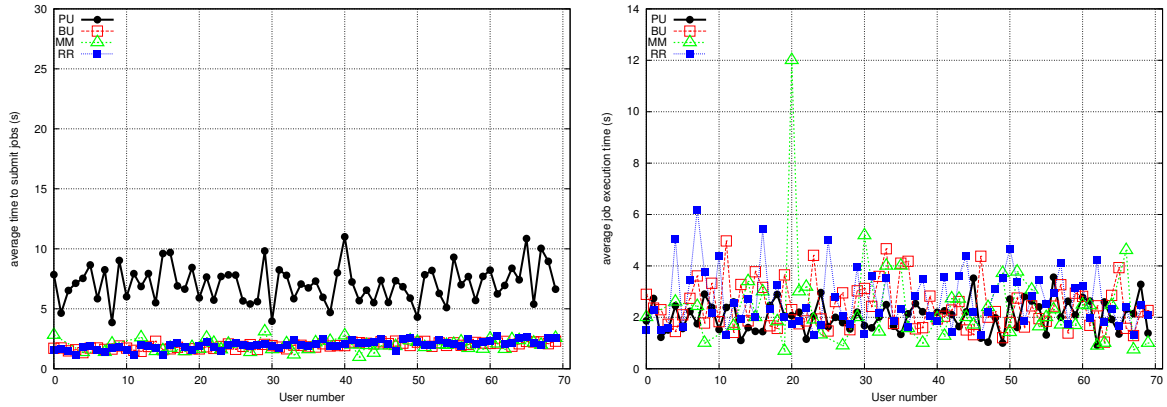

Figure A.7: Jobs per user 5: average time to submit and average execution time

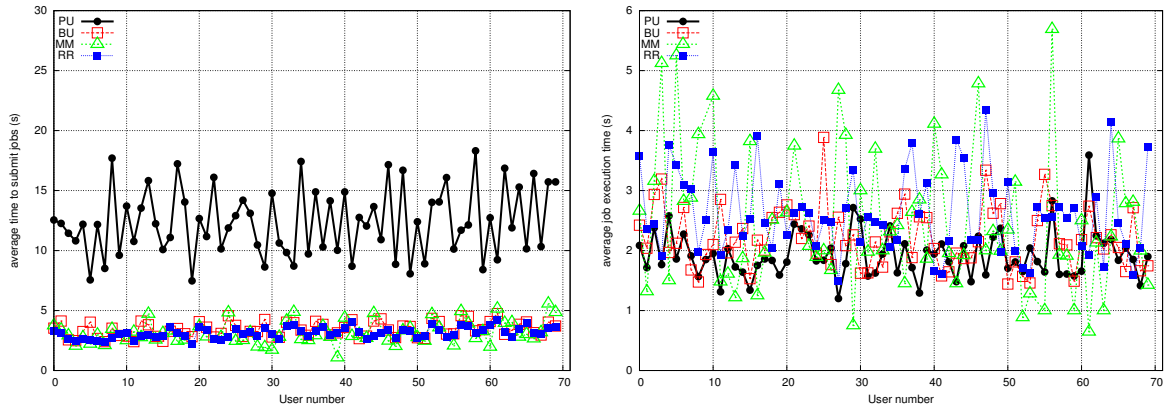

Figure A.8: Jobs per use 10r: average time to submit and average execution time

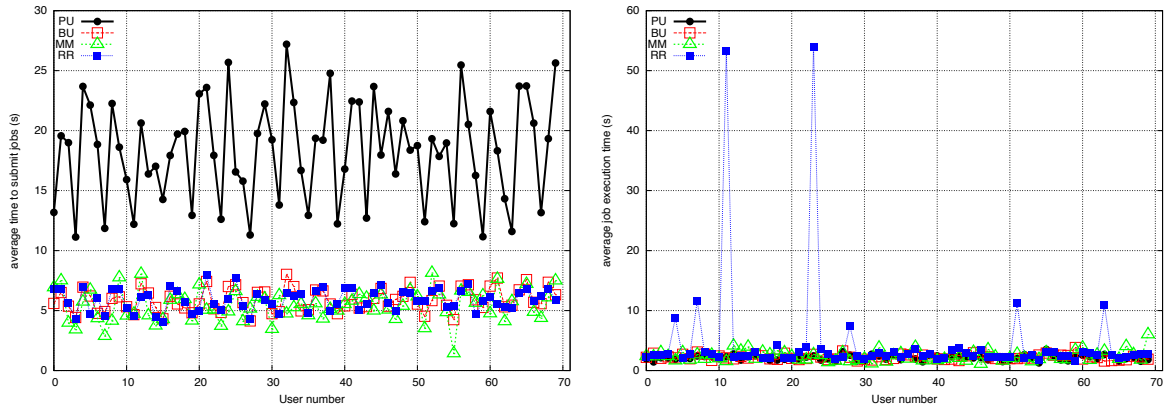

Figure A.9: Jobs per user 20: average time to submit and average execution time

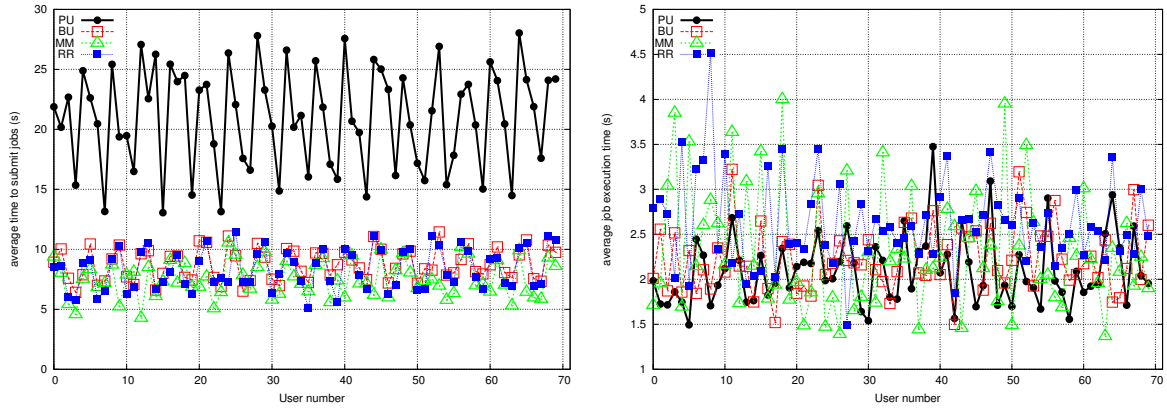

Figure A.10: Jobs per user 30: average time to submit and average execution time

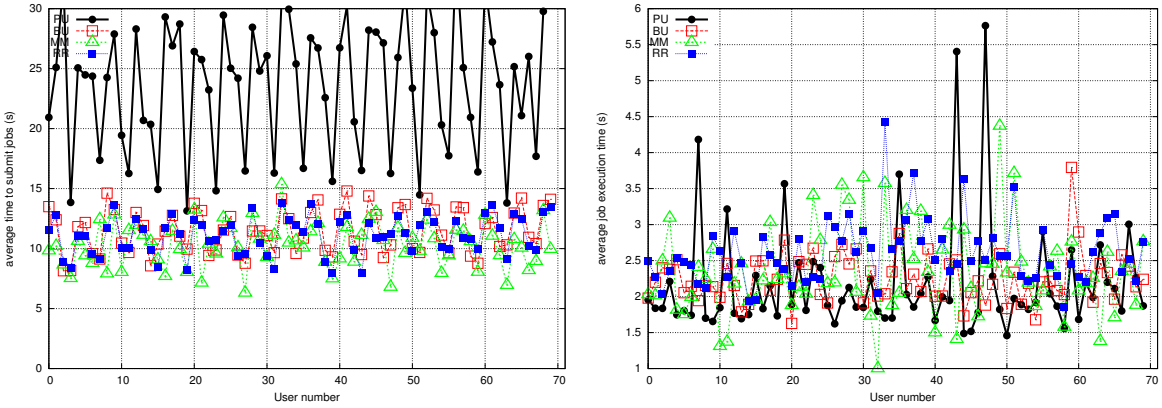

Figure A.11: Jobs per user 40: average time to submit and average execution time

A.2.2 Metrics -user's average utility and job success ratio

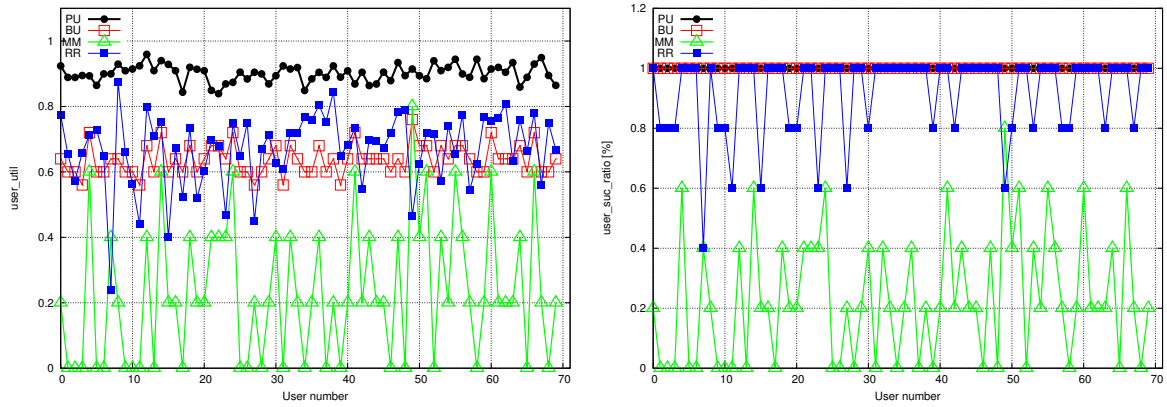

Figure A.12: Jobs per user 5: user's average utility and job success ratio

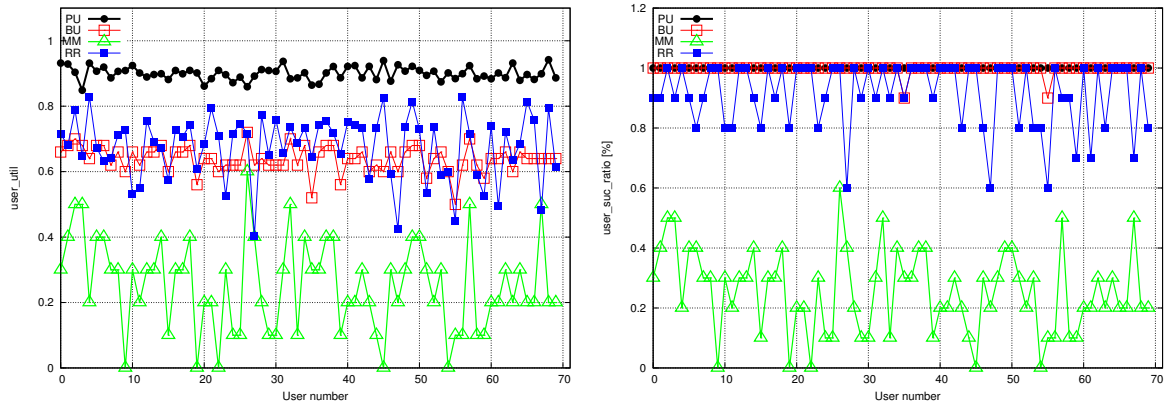

Figure A.13: Jobs per user 20: user's average utility and job success ratio

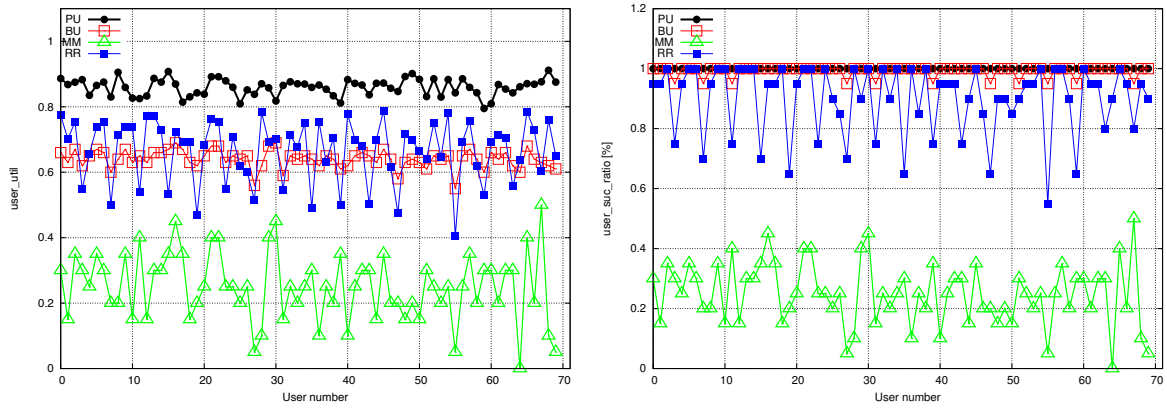

Figure A.14: Jobs per user 20: user's average utility and job success ratio

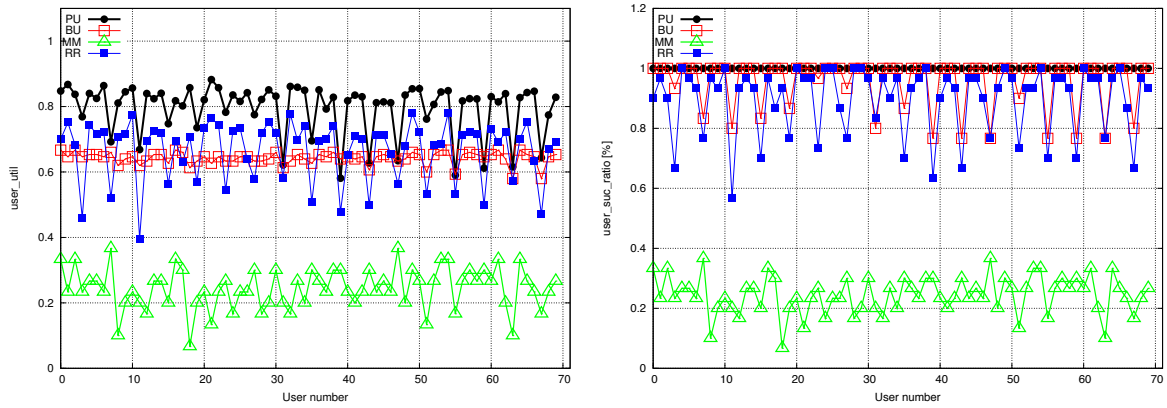

Figure A.15: Jobs per user 30: user's average utility and job success ratio

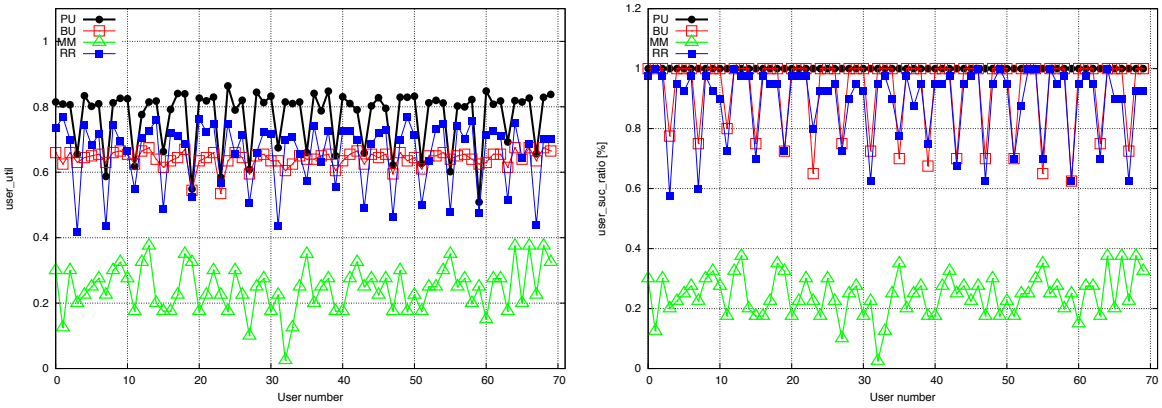

Figure A.16: Jobs per user 40: user's average utility and job success ratio

A.3 Job inter-departure rate

A.3.1 Metrics - job submission and execution time

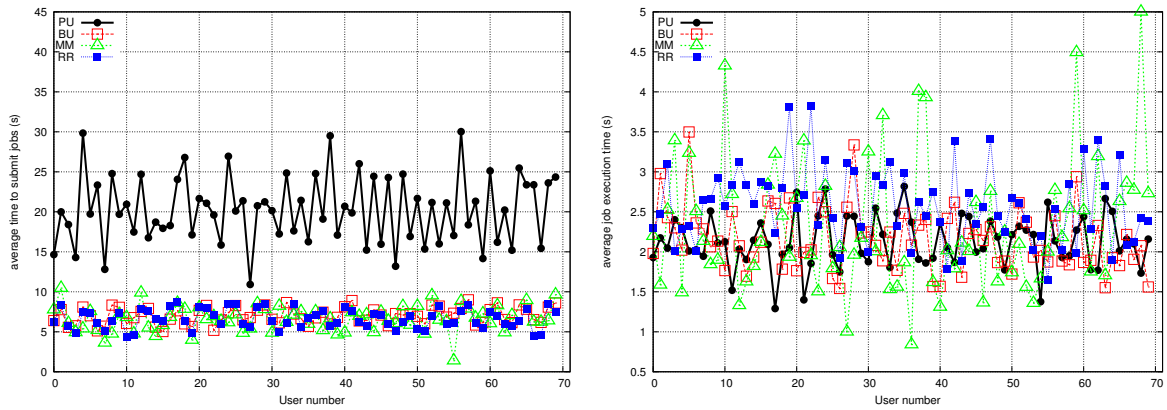

Figure A.17: Poisson mean 0.75: average time to submit and average execution time

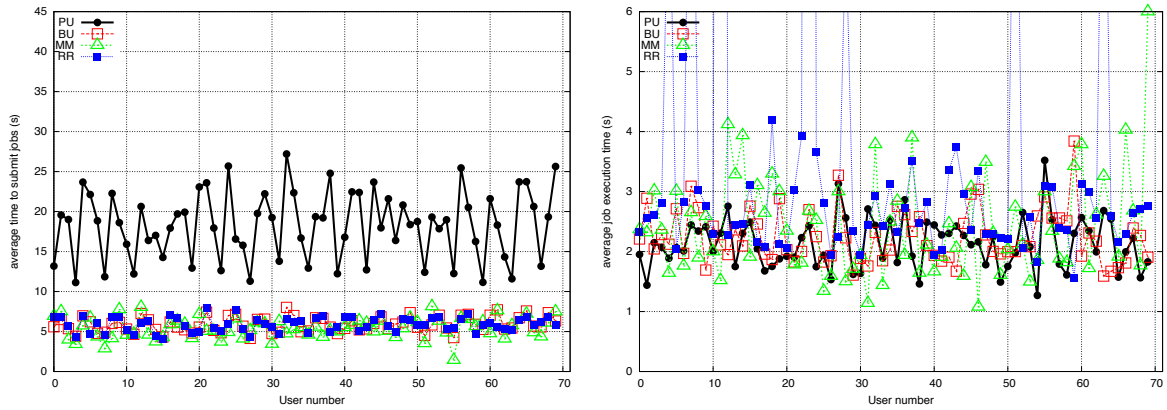

Figure A.18: Poisson mean 1: average time to submit and average execution time

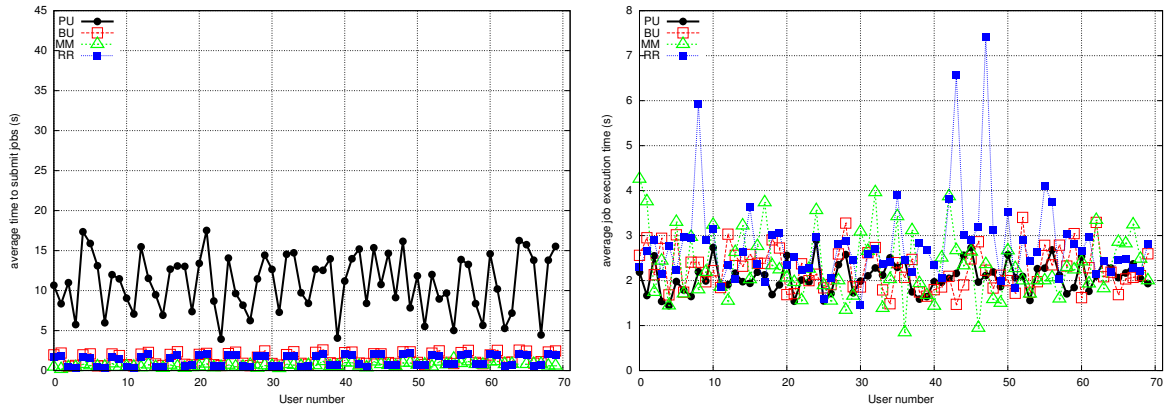

Figure A.19: Poisson mean 2: average time to submit and average execution time

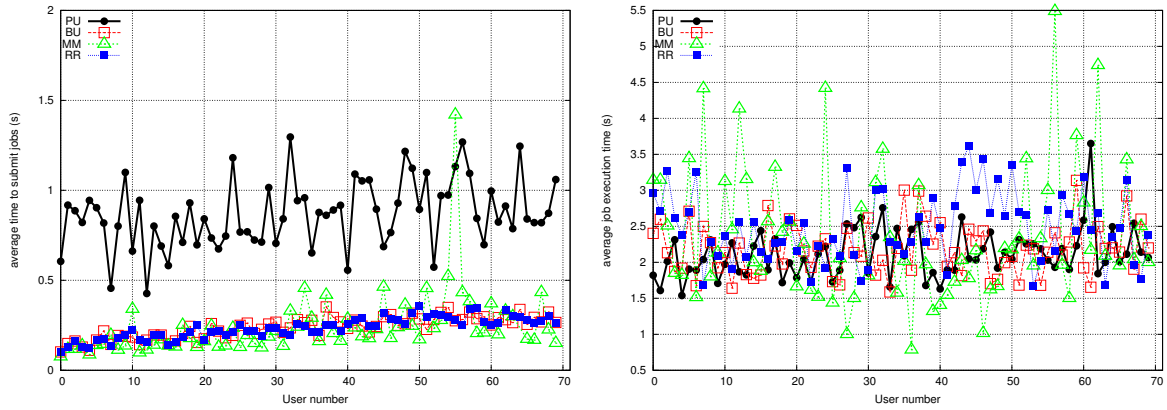

Figure A.20: Poisson mean 4: average time to submit and average execution time

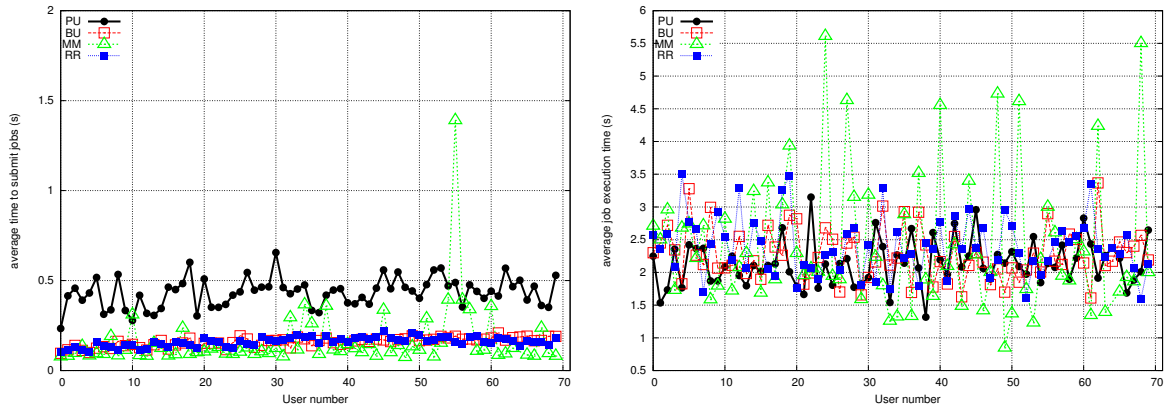

Figure A.21: Poisson mean 8: average time to submit and average execution time

A.3.2 Metrics -user's average utility and job success ratio

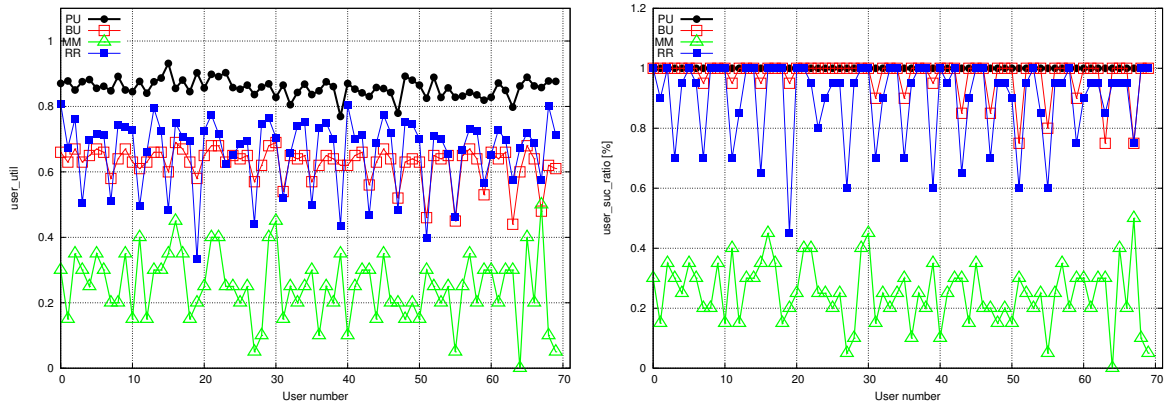

Figure A.22: Poisson mean 0.75: user's average utility and job success ratio

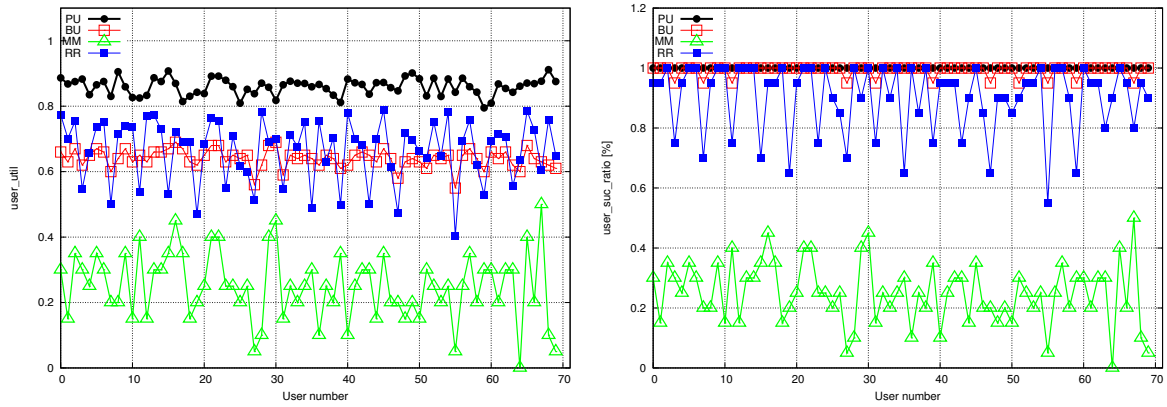

Figure A.23: Poisson mean 1: user's average utility and job success ratio

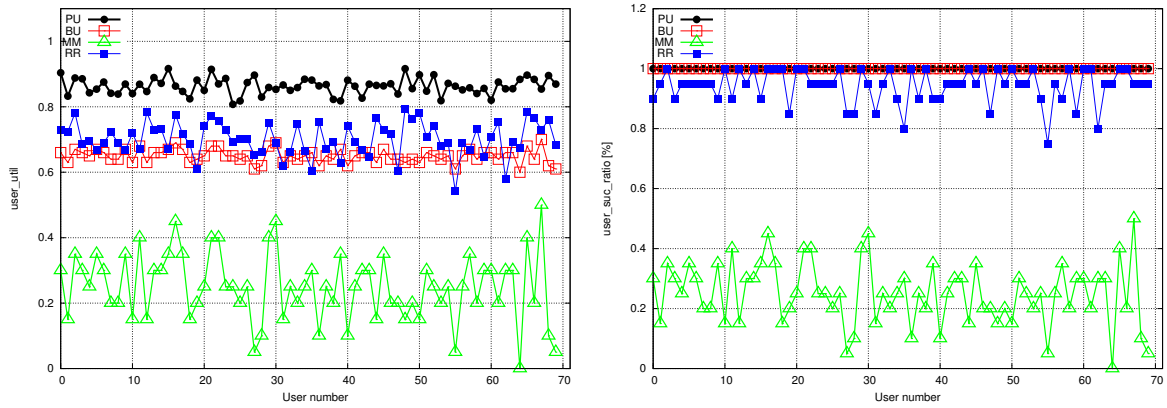

Figure A.24: Poisson mean 2: user's average utility and job success ratio

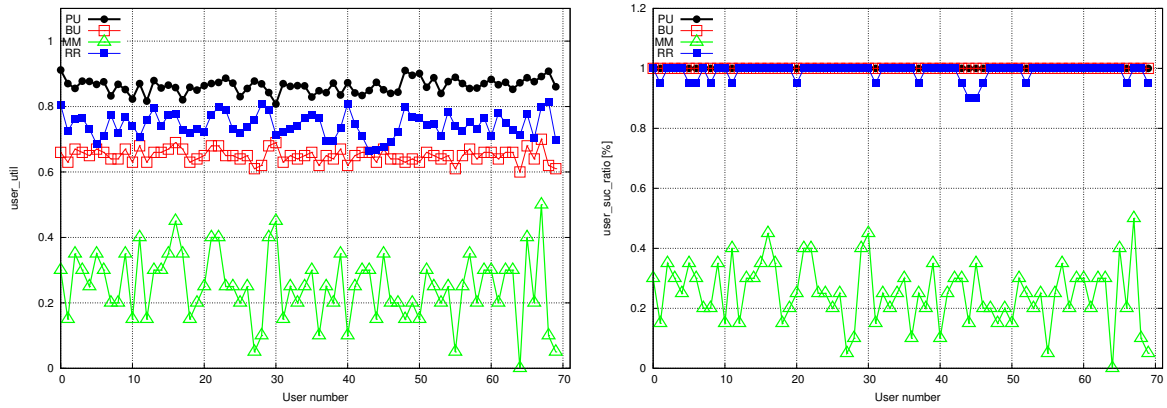

Figure A.25: Poisson mean 4: user's average utility and job success ratio

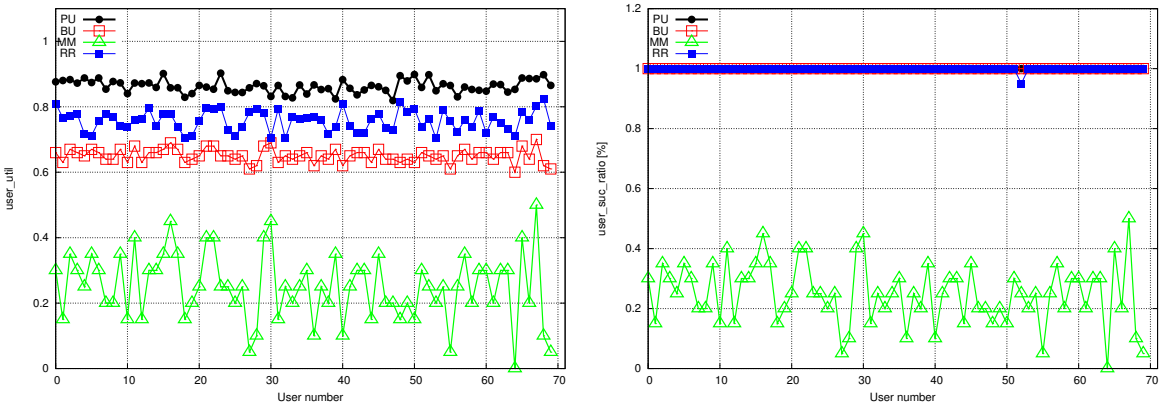

Figure A.26: Poisson mean8: user's average utility and job success ratio