Adaptive Orchestration of Resources in Distributed Wide Area Large Scale Infrastructures

By

John Alexander Sanabria Ordoñez

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
In
COMPUTING AND INFORMATION SCIENCES AND ENGINEERING

University of Puerto Rico
Mayagüez Campus
2009

Approved by:

Jaime Seguel, Ph.D. Member, Graduate Committee	Date
Domingo Rodríguez, Ph.D. Member, Graduate Committee	Date
Manuel Rodríguez, Ph.D. Member, Graduate Committee	Date
Wilson Rivera, Ph.D. President, Graduate Committee	Date
Pedro Resto, Ph.D. Graduate School Representative	Date
Nestor Rodríguez Rivera, Ph.D. Chairperson of the Program	Date

ABSTRACT

Adaptive Orchestration of Resources in Distributed Wide Area Large Scale Infrastructures

By

John Alexander Sanabria Ordoñez

The goal of this thesis is to take a step to understanding the resource management of massively and scattered distributed systems. A framework to predict grid resources behavior and leverage the execution of long running tasks over computational grids has been developed. This framework employs statistical analysis for estimating the resource behavior and uses a divisible load approach to increase the throughput and reduce the idle time exhibited by computational resources. The proposed approach focuses on an opportunistic pull resource selection mechanism: a number of very light agents are deployed in nonintrusive way running in a user space. Initially the framework collects information on user requirements and application deployment, assigns a subset of jobs to available resources, and periodically the selected pool of resources is updated to opportunistically choose the resources that better complete the assigned jobs. The statistical analysis process evaluates in run time different probabilistic functions to determine the one that better model a resource behavior. Experimental results show a significant reduction of the application makespan along with good estimations of the resource behavior.

RESUMEN

Adaptive Orchestration of Resources in Distributed Wide Area Large Scale Infrastructures

Por

John Alexander Sanabria Ordoñez

El objetivo de esta tesis está dirigido a comprender la administración de recursos sobre sistemas distribuidos masivos y globalmente dispersos. Se ha desarrollado un marco de trabajo que predice el comportamiento de recursos grid y apoya la ejecución de tareas de larga duración sobre grids computacionales. La aproximación propuesta se enfoca en un mecanismo de selección oportunista de recursos: un número de agentes livianos se desplegaron de manera no-invasiva en espacio de usuario. Inicialmente el marco de trabajo recoge información sobre los requerimientos del usuario y del despliegue de la aplicación, asigna un subconjunto de tareas a los recursos disponibles, y periodicamente el grupo de recursos seleccionados es actualizado de modo que se escoja de manera oportunista los recursos que mejor completen los trabajos asignados. El análisis estadístico evalua en tiempo de ejecución diferentes funciones probabilisticas para determinar aquella función que mejor modela el comportamiento de un recurso. Resultados experimentales muestran una reducción significativa en el tiempo de ejecución de la aplicación junto con una buena estimación del comportamiento del recurso.

To my lovely parents...

ACKNOWLEDGMENTS

This work was partially supported by the NSF CISE-CNS Grant No. 0424546 under the WALSAIP (Wide Area Large Scale Automated Information Processing) Project.

First I would like to thank God because He has always put all the things in place and has taken special care of me. To my parents because they have set a great example and have been fundamental support underlying all my achievements. To Drs. Wilson Rivera and Domingo Rodriguez for their continuous support. I want also to thank the other members of my graduate committee Drs. Jaime Seguel and Manuel Rodrguiez who provided valuable comments. To my family in Christ: the members of Primera Iglesia Bautista de Mayaguez and Primera Iglesia Bautista de Palmira, all my relatives and friends for their prayers and everyone who has contributed to this work in some way: encouragers, readers and reviewers (Pablo, Kennie, Sarah, Alida, Nestor, Maria Patricia, Bob, Eileen). Special thanks to Susana because she taught me to see life from another perspective. I am grateful for her company, her support, her patience and the great moments that she shared with me. Lastly, I thank the Puerto Rican people who were so kind to me. God bless all of you.

John A. Sanabria
Mayaguez, Puerto Rico
June 2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivating Scenario

There have been tremendous technological advances in computer and networking technologies during the last decades. Those advances are demanding new theories, methods and techniques to integrate effectively heterogeneous resources including computational nodes and storage units.

During the mid-nineties multi-computer systems built from commodity components drew the attention of system managers and the research community. At National Aeronautics Space Administration (NASA) Goddard Space Fligh Center a group of researchers developed a Linux driver for Ethernet network cards and built the first cluster using commodity-off-the-shelf components. Clusters are now widely deployed in education and industrial sectors. They are characterized by hardware homogeneity, low levels of failure and well known management policies since they are governed by a single administrative domain. Since then clusters are an important participant on the high performance computing landscape. In fact, the Top 500 Supercomputers site offers interesting statistical results. Eighty two percent of the 500 super computers are clusters. Over seventy percent (73.80%) of them employ Intel EM64T technologies, e.g. Intel Core 2 and Intel Centrino, 87.80% use Linux as their operating system and 56.40% are interconnected through Gigabit Ethernet.

In the late nineties Grid Computing emerged as a well-founded alternative for distributed computing. Grid infrastructures embrace not only traditional computational systems but also

specialized devices such as simulators, particle accelerators, and telescopes, among others. Grid Computing mentions the relevance of ubiquity computing not only from the point of view of the end-user but also from the point of view of the infrastructure. Grid Computing offers dynamic infrastructures created on user demands and built from scattered resources governed by multiple administrative domains. Those infrastructures are known as Virtual Organizations (VOs). The computational assets are now presented as services. The service concept provides a higher level of abstraction where service users can focus on functionality and capability of the resources, instead of the technical details.

Grid Computing environments present additional challenges at the infrastructure, system and functional levels. Grids inherit the well-studied problems found in classical distributed systems, such as synchronization, security, scalability, and monitoring, among others. Moreover, these problems are often more complex in Grid systems than legacy distributed systems because new factors, such as the number of resources engaged, diverse resource management policies, the uncertainty of resource availability and network variable latency could affect the normal execution of the system. There is thus a heightened interest among system developers and users in efficient resource orchestration employing on-demand provisioning techniques.

One important challenge in such environments is *resource selection*: For a given application and a heterogeneous pool of computing, communication, and storage resources; decide dynamically which set of virtual resources should be assigned to each application component, such that it mantains adequate quality of service. This is a difficult problem due to multiple factors. First, the performance and availability of grid resources varies over time. Compute nodes are heterogeneous and exhibit different capabilities, the quality of network connections between resources varies, resources may become overloaded, unavailable because of failures, or even new resources may become available. Second, resources may be under different administrative domains. Users have no control on the policies ruling the use of those resources and furthermore it is difficult to determine a priori which resources are more suitable for a particular application.

In order to maintain a reasonable performance level, it is desirable that resources selection

mechanisms adapt to the changing conditions. The resource selection can be repeated during application execution either at regular intervals or when performance degradation is detected. The system model, in this work, is a federation of distributed clusters. Each cluster is located in a different administrative domain, and controlled by a local resource manager. In a higher level clusters are interconnected in a wide area network and grid resource brokers are responsible for coordinating the resources. Resource brokers may have different architectures. There can be a central resource broker which is responsible for job scheduling in the whole Grid. On the other hand, every single user can have her own broker that makes decisions on her behalf. We believe that the latter approach provides more scalability and also allows the implementation of self-adaptive opportunistic resource allocation mechanisms.

1.2 Problem Statement

We propose to study the following Resource Assignment Problem. For a given application and a heterogeneous pool of resources, decide which set of physical resources should be assigned to each application component, such that it mantains service level agreements under uncertainty environment conditions.

We seek to establish a method for monitoring computational resources, modeling the behavior of the resources through probability functions thatexhibit the residence times in various job stages, and using of these statistical models to produce rankings of resources according to various performance metrics. The result of this work must be an architecture that integrates these components in addition to the corresponding benchmarking.

1.3 Problem justification

Legacy applications keep driven critical processes on industry and education sectors. Thus, there is an important quantity of legacy code running critical procedures in which response times increase as much as the amount of data required to be processed grows. Legacy applications that receive as input, the data to be processed and parameters to delimit the action range can be easily treated as parameter sweep applications (PSA). PSA are characterized because they tackle problems to exhibit high granularity. Thus, big input data is divided in smaller chunks

and the pieces are delivered to different computational nodes in such a way that the total execution time decreases.

Legacy applications presenting a PSA like-structure, are suitable for being executed on loosely coupled infrastructures. Many times, those infrastructrues are built by resources scattered geographically which are interconnected through public networks to create an environment where network links observe high network latencies and availability and reliability uncertainty patterns. Since PSA response time does not depend primarily on the quality of communication links but the computation capacity of the resources, computational grids are an excellent option as computational platform for PSA. Problems on research areas such as data mining, signal processing and sequencing, among others; could take advantage of this kind of computational platform and applications. For instance, Google employs the Map-Reduce paradigm for empowering its searches. Map-Reduce divides an input data amongst the available resources (Map) and later collect the response from the aforementioned resources (Reduce). However, for achieving quickly response times, the paradigm relies on a high performance distributed filesystem to efficiently execute the Reduce operation even on low profile network links.

Figure 1.1. PRAGMA members distribution

There are several international, regional and national grid initiatives around the world. These initiatives, many times, embrace powerful computational resources (aka. clusters) to include hundreds of computational nodes per resource. The Pacific Rim Assembly Grid Middleware Application (PRAGMA) is an initiative which aims to integrate research and education centers located around the Pacific area, Figure 1.1. Those institutions are willing to share their computational resources amongst the authorized PRAGMA users. PRAGMA as well as other grid platforms formed by resources cluster-based resources, present an excellent asset for executing parameter sweep legacy applications. There is an important number of legacy applications to be executed on grid infrastructures. However, a computational framework is required to exploit the unsteady characteristics of the resources and take advantage of the application requirements in such a way to generate efficient execution plans.

Despite grid-enabled resources expose their caracteristics through standarized interfaces, these interfaces barely reflect the management policies that govern the resources behavior. We claim that there is a lack of services to standarize the definition and exposition of management policies to govern resources. This fact could be due to the resource managers have a design resource oriented. This design focuses on handling the resources according to constraints defined over resources characteristics per se, for instance CPU speed, storage capacity and architecture, among others. User oriented resource managers are more difficult to implement because user constraints and requirements could be specified not only in functional but also in non-functional terms. For instance, for a user the resource selection could be driven not only by a deadline or budget but also by security or politic issues, such as taking into account those resources whose security level is Evaluation Assurance Level 4(EAL4) or higher. Therefore, the computational platform would treat the grid resources as observable systems. By employing direct observation over application executions and getting information from monitoring tools, the platform would create an internal view or snapshot of the grid environment. Using forecasting techniques, the platform also must be able to estimate and predict resource and application performance considering unsettled conditions observed in the environment.

1.4 Contributions

The main contributions of this research work are summarized as follows.

- *Computational Framework for Grid Environments*

  We describe the design and implementation of Gridjobs, a grid computing enabling framework to provide functionalities for deployment, integration and management of grid computing infrastructures. This framework employs statistical analysis over historical data along with self-adaptive mechanisms to allocate efficiently task to resources. We demonstrate the utility of this tool in a real large scale environment using a computationally intensive application. A cycle short time Fourier transform (CSTFT) kernel is deployed and executed in distributed resources pertaining to the Pacific Rim Applications and Grid Middleware Assembly (PRAGMA) infrastructure.

- *Adaptive Scheduling*

  We evaluate different ranking policies based on the statistical information from cluster's historical performance. We show that the design of the framework allows plugging different ranking policies focusing on estimated time execution times and estimated failure probability without requiring application performance models. Our approach focuses on an opportunistic pull resource selection mechanism: a number of agents are deployed in non-intrusive way running in a user-space. Initially the framework collects information on user requirements and application deployment, assigns a subset of jobs to available resources, and periodically the selected pool of resources is updated to opportunistically choose the resources that better complete the assigned jobs.

1.5 Summary

Distributed computing has evolved toward computational infrastructures built from comodity hardware and open source software tools. Those environments have reached an important maturity level that allows to compete against proprietary solutions on the high performance computing field. Due to significant connectivity improvements at regional, national and international level; distributed computing is now tackling the problem of orchestrating several

computational resources governed by different administrative domains and interconnected via public networks.

Grid infrastructures scattered geographically exhibit significant uncertainty levels because the absence of mechanisms to support QoS. Resources performance could not be a priori determined since multiple factors could affect the resources availability and accessibility. Thus, adaptative approaches are required for maintainig acceptable levels of reliability.

In this chapter, we stated the problem of resource management in distributed wide area large scale infrastructures, and established the scope of our research. we finally, summarize the contributions of this thesis.

CHAPTER 2

Background and Related Work

2.1 Grid Computing Environments

Grid computing [1] relies on standard protocols and open technologies in order to provide non-trivial quality of service through the orchestration of Grid enabled resources. Open technological solutions along with non-proprietary protocols are required to avoid excluding resources because of licenses or constraints imposed by hardware and software manufacturers. [2]

Grid leverages the integration concept to a higher level, as Grid infrastructures are suitable to incorporate not only processing units and storage devices but also sensing instrumentation and specialized equipment. Grid wraps all those components as Grid Services. Using Web Service Description Language (WSDL) the functionality of hardware and software components is encapsulated in such a way that integration is possible. This conceptualization renders an appropriate computational infrastructure that could be composed by services from different service providers which could be scattered geographically. Grid Services enable composition of distributed components. This composition requires the incorporation of sophisticated orchestration mechanisms to support lifetime resource management and notification primitives. All these complex interactions demand reliable invocation, single sign-on, authorization and delegation services that allow an even and trustable interaction between requesters and providers. [3]

The Grid concept lacks of a unique and widely accepted definition, amongst experts and users. Despite a common ground of features and characteristics are already identified and

required in order to label a system as Grid. Specifically, a Grid must provide facilities for collaboration and aggregation as well as support for hardware and software heterogeneity, decentralized control, access transparency, scalability, reconfigurability and security mechanisms [4]. Nowadays, there are various implementations of the Grid concept. P. Asadzadeh et. al.[5] compares four important projects: Legion, UNICORE, Globus and Gridbus. Although all of these middlewares provide the basic building blocks to create operational Grid systems, they exhibit differences on implementation technologies, runtime platforms, and distribution model.

Grid systems can be categorized as computational oriented, data oriented and service oriented. Most of the implementations are computational oriented. Some of the most relevant Grid implementations are presented in the following section. Application Level Scheduling (AppLeS) [6] is a computational and grid oriented application. AppLeS consists of a group of agents developed for individual applications on production grids. It relies heavily on Network Weather Service (NWS) for monitoring and predicting resource performance. AppLeS could also interact with other systems such as Globus, Legion and NetSolve. AppLeS provides agent templates that can be extended in such a way that applications following common parallel and distributed programming paradigms such as master/slave or parameter sweep application could be easily deployed. Despite scheduling is centralized, the execution is decentralized and carried out by local resource managers.

Condor [7], [8] is a system to scavenge for PCs, workstations and clusters that belong to different administrative domains. It provides a heterogeneous computational environment harnessing idle CPU cycles. Condor supports checkpointing and migration between different computational environments. Resource requests and offers are described in the Condor classified advertisement (ClassAd) language. The ClassAd language includes a query language as part of the data model, allowing advertising agents to specify their compatibility by including constraints in their resource offers and requests. Condor exhibits characteristics of a computational grid with a flat organization. With no support for quality of services, it presents a centralized scheme for query executions as well a centralized scheduler to suggest scalability problems for addressing mid- and large-size systems.

Globus [9] presents a virtualized view of geographically distributed resources to the applications running on top of it. The Globus ecosystem consists of multiple services such as security, resource location, resource management, data management, resource reservation and communications. From a layered architecture, Globus provides a hierarchichal integration of Grid components and services. This feature encourages the utilization of low level services to leverage the composition of more sophisticated services. Globus does not support scheduling policies but delegates policy implementation to third party local resource managers.

Finally, gLite[10] is another important implementation of Grid which presents remarkable similarities with Globus at functional level. It provides a framework for building grid applications tapping into the power of distributed computing and storage resources across the Internet. gLite is supported by the Enabling Grids for E-sciencE(EGEE) project. gLite exhibits a modular design that allows the installation of components that a resource requires. In contrast, Globus presents a monolithic approach that demand all components must be installed on a Globus based Grid resource.

2.2 Grid Systems Modeling

Different approaches to model Grid Systems exist. At a structural level, diverse graph approaches have been used to model distributed scenarios and systems. Directed Acyclic Graphs (DAG) are used to model a chain of tasks ordered by their execution dependencies. The nodes in a DAG correspond to tasks and directed edges represent precedence relationship among tasks. Kwok and Ahmad [11], for example, use extensively DAGs to model scheduling and mapping. The scheduling and mapping problem requires the allocation of multiple interacting tasks of a single application in order to minimize the completion time on the parallel computer system. Consequently, the objective of the DAG scheduling is to minimize the overall program finish-time by proper allocation of the tasks to the processors and arrangement of execution sequencing of the tasks. There are a number of variations in the generic DAG model that can be found elsewhere in the literature. Although suitable to model task dependences and dynamics of tasks execution, DAGs do not provide enough representative features to model complex interaction of distributed services and uncertain network conditions.

Petri Nets, on the other hand, can be used as mathematical representations of discrete distributed systems. As a modeling language, it graphically depicts the structure of a distributed system as a directed bipartite graph with annotations. As such, a Petri net has place nodes, transition nodes, and directed arcs connecting places with transitions. Murata [12] provides a complete description of Petri Nets theory. Colored Petri Nets are useful to get more compact models with a higher level of abstraction, especially when the system can be expressed as a set of components with similar behavior. Temporized Petri Nets allows modeling discrete event systems, manufacturing processes, and distributed systems using probability distributions to fire transitions. Petri Nets allows for modeling dynamic transition of task and discrete events in a distributed system, but similar to DAGs, do not provide enough representation to model adaptive functionalities.

Initiatives directed toward modeling the load of Grid Systems are have been also developed. X. Zhang et. al. [13] uses data mining techniques on logging and bookkeping files for identifying the load behavior of around 20,000 CPUs to handle 20,000 jobs on average. Logging and bookkeeping files are generated by a grid broker, which is a component of gLite in charge of managing jobs in an EGEE architecture. The information found on these files is partitioned according to users who submit the jobs per week. From each subset a consistent snapshot of grid use is expected. Then, discriminant learning is run over each subset in such a way that a hypotheses discriminating good from bad jobs in the subset are determined. The hypothesis are now used as new features used to redescribe the jobs in the test set. Double clustering of the hypothesis and the jobs is reached and the significance of the clusters is estimated comparing the user-based and week-based hypothesis.

Other works focus their attention on programming models. Those models are essential because they provide a set of primitives, operations, functions and data types to present the ground necessary to develop computational solutions. They provide a simplified view of the machine architecture where they are executed. Note that more relevant programming models have been selected due to non-functional properties. For single computers, object-oriented and component-based paradigms are the preferred programming models. For parallel computers,

message-passing paradigm is selected. Although the later model barely represents the machine architecture, it exhibits the closest match between machine architecture and programming model, leading to efficient program execution. Kielmann et. al. [14] consider at least five kind of applications such as legacy code, parallel applications, grid-aware codes, support tools and service and resource management. Each type requires a particular view of the Grid. Some of them require a high level abstraction of the system (Legacy codes and parallel applications) but others low or no level of abstraction (support tools and service and resource management). From this point of view, there is not a single programming model but many; in such a way that each model best fits the functional properties of its applications.

Amin et. al. [15] present a framework to close the existing gap between grid infrastructures and applications. This application programming interface (API) tackles problems such as interoperability amongst multiple Grid backends, code re-usability for rapid prototyping and extensible architecture that allows the collective and incremental development. This programming model works around of three patterns that represent different kind of tasks available in Grid systems such as single tasks, tasks with dependencies and workflows.

Finally, Németh and Sunderam [16] presents a formal approach isolated from grid implementations but used to define basic characteristics present in grid systems. The proposed model does not intend to be a final product but a starting model that allows the derivation of more refined models. This model is built from scratch because it does not focus on compatibilities with available protocols or systems but in functionalities that grid systems must provide. Using a declarative approach, this model specifies how to realize or decompose a given functionality instead of what it must provide. In addition, it adopts an architectural/system developer's point of view. The proposed model follows the nomenclature provided by Abstract State Machine(ASM). Then, diverse universes are defined, such as $APPLICATION, USER, PROCESS, NODE$ and $RESOURCE$, among others. Along with the universes, several functions with different arities are specified. For instance, *mapped*: $PROCESS \rightarrow NODE$ and *belongsTo*:$RESOURCE \times NDOE \rightarrow \{true, false\}$; which express where a process was mapped and if a given node belongs to a given resource, respectively.

From the aforementioned universes and functions, the model establishes rules to represent different case uses such as communication, resource selection, delegation and mapping, among others. Finally, a refinement of the initial model is provided in such a way that more algorithmic details of the aforementioned rules can be given.

In [17], Németh and Sunderam refine their initial work and apply their proposed model not only against conventional distributed systems but also against grid implementations such as Globus and Legion. The paper presents a methodology to determine if a distributed system is a grid system or not. In the same way, the model is used to evaluate some other computational systems to handle a large set of resources such as SETI, Condor, OGSA and SGE. The former three are considered grid systems. Despite showing as a single user, SETI harnesses several computational resources that belong to different owners. Condor supports matchmaking mechanisms for mapping resources and jobs. In addition, a demanding job could employ resources from other pools creating virtual organizations on-the-fly. On the other hand, SGE is not compliant with grid characteristics defined in the model because the number of resources and users are known beforehand.

2.3 Divisible Load Theory

Divisible load theory (DLT) offers a tractable approach to scheduling problems considering computation and commmunication characteristics of parallel and distributed systems. DLT is adequate for applications which are compounded by large sets of independent tasks with low granularity. Robertazzi [18] describes ten reasons why DLT is a suitable approach to the scheduling problem. DLT assumes that processor and links exhibit linear behavior. "Setting up a continous-variable model and assuming that all processors stop computing at the same instant lets you determine the optimal amount of total load to assign to each processor or link".

To illustrate the DLT approach assume a star network topology with a root processor P_0 is processing some load itself while simultaneously distributing loads to processors P_1 through P_4. Let α_i be the size of data chunk assigned to each processor, w_i the computing speed of the ith processor, and z_i the transmission speed of the ith link. Then, $\alpha_i w_i$ is the processing

time of the ith processor, and $\alpha_i z_i$ is the transmision time of data over the ith link. Figure 2.1 depicts a processing scenario where the load transmission for all nodes starts at the same time and its completion time is equal.

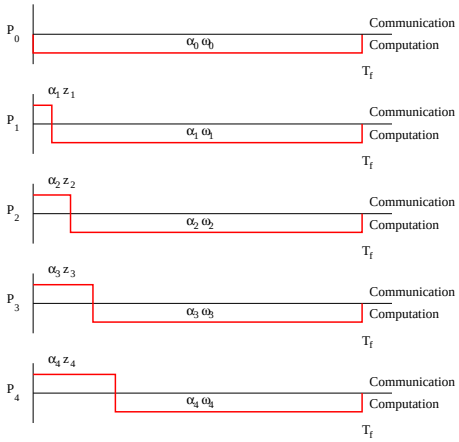

Figure 2.1. Gannt diagram for star network topology.

Using recurrence equations, we have:

$$\alpha_i = \left(\frac{z_{i-1} + w_{i-1}}{z_i + w_i}\right)\alpha_{i-1} \tag{2.1}$$

$$= f_{i-1}\alpha_{i-1} = \left(\prod_{j=1}^{i-1} f_j\right)\alpha_1 \qquad i = 2, 3, 4 \tag{2.2}$$

For processor P_0:

$$\alpha_0 = \left(\frac{z_1 + w_1}{w_0}\right)\alpha_1 = \left(\frac{1}{k_0}\right)\alpha_1$$

Since $\sum_{i=1}^{4} \alpha_i = 1$, and assuming that all processors have the same computing and communication speed, a simple solution is given by:

$$T_f = \alpha_0 w_0 = \left(\frac{1}{4 + 1/k_0}\right)(z + w)$$

From the basic DLT model, other models have been derived for different network topologies, such as linear daisy chains, trees, buses, hypercubes,and two- and three-dimensional

meshes. DLT also provides a methodology to derivate single network elements which are equivalent to complex network infrastructures. Novel approaches are also incorporating monetary accounting in order to find optimal execution plans considering payments to computer owners. DLT is an active field of research in diverse areas. For instance, in the formal field, some researchers are extrapolating the model to more complex network infrastructures like hypercubes. On the technological field, DLT models must be integrated with data-parallel languages such as high-performance Fortran and high-performance C++.

More representative works to conjugate DLT with Cluster/Grid systems are now discussed. Beaumont et. al. [19] present a unified theoretical perspective to summarize previous works and addresses new questions to foster the research of new DLT-based approaches. This work only considers star and tree network topologies where tasks are submitted in one-round or multiround basis. The proposed one-round algorithms tackle issues such as selection and ordering of the workers, and size of the chunks. Multiround algorithms determine chunk sizes per round and number of rounds. The approaches presented in this work also consider affine costs in such a way that a more realistic representation of real computational scenarios is possible. However, those approaches do not focus on the characteristics exhibited for more complex scenarios like Grid systems, for instance resource discovery, authentication, delegation and so forth.

Viswanathan et. al. [20] propose a elaborated computational model for cluster/grid systems. In this work, the problem of computational nodes with a finite buffer is considered and the model proposed also integrates other factors to refine the representation of real distributed systems such as load variation over time, deadline constraints and control policies. In addition, a novel algorithm which involves multiple computational elements for getting a scheduling plan is also especified. That said, Coordination Node (CN) determines nodes participanting in the computation and scheduling parameters. It sends the information to the selected nodes. The selected nodes estimate the buffer availability and send the information to the CN. Finally, CN distributes suitable chunk size to the compute nodes.

Cardinale and Casanova [21] develop a modified version of the basic DLT model which

minimizes the average steady-state job turnaround time. They quantify the amount of global information required to achieve efficient scheduling. Using simulations with different application and platform scenarios, they have found that a global information is only needed under high workloads. In that sense, by considering scenarios with high workloads and using dynamic information, the performance is improved by around 10% when compared with static information. Some intuitive results were achieved, for example when the communication and computation ratio is high then it is worthless to distributed the load, and a scheduling plan to divide the load equally amongst the resource participants exhibits the worst turnaround times.

2.4 Resource Management

Krauter et. al. [22] presents a comprehensive taxonomy to describe different management architectures. This taxonomy was derived from a study of different architectures and functionalities exhibited by different resource management systems. Different taxonomies are defined according to machine organization, resource management and scheduling characteristics. The machine organization taxonomy identifies three categories Flat, Cells and Hierarchical. These categories are determined according to how the machines interact and if they are grouped or not. The resource taxonomy defines different categories according to the interfaces employed for interacting with other Grid components. Thus, five categories were determined: resource namespace organization, quality of service support, resource information store organization and resource discovery and dissemination. The scheduling taxonomy considers the way the schedulers are organized, how they estimate the state of resources, rescheduling policies, and scheduling policies. For instance, scheduling policies coud be categorized as fixed and extensible. The fixe category could also be subdivided into system oriented and application oriented. The former is widely adopted and only considers resources requirements. The latter, considers application requirements and tries to optimize specific metrics such as completion time.

Buyya et. al. [23] identify issues in resource management and scheduling considering economy factors under grid environments. The approach is not only applied to grid environments but also applied to federated clusters or hyperclusters. This work remarks the importance of economic models in the Grid environments. In that sense, current Grid implementations such

as Globus, Legion, NetSolve, Ninf, AppLeS and Nimrod/G, among others, do not consider economy of computations as fundamental issue in Grid. Thus, a GRid Architecture for Computational Economy (GRACE) middleware has been proposed. It interoperates with Globus and Nimrod/G systems. GRACE takes advantage of deadline and cost-based scheduling mechanisms provided by Nimrod/G and extend them in order to include economy of computations ideas. In addition to existing Grid components such as Grid Resource Broker, Grid Middleware and Local Resource Manager, the GRACE middleware adds a Grid Trade Manager and Trade Server. GRACE interoperates with existing Grid through a set of protocols and APIs defined for the aforementioned components.

K. Lai et. al. [24] proposes a tool for resource allocation named Tycoon. Tycoon implements an Auction Share scheduling algorithm to provide a market based distributed resource allocation system. This work compares resource allocation systems based on economic efficiency, utilization, risk and fairness. Tycoon separates the allocation mechanism from agent strategies. This separation allows great flexibility at installation application time because the user would specify functional and non-functional application requirements and simplifies, at the same time, the development of allocation mechanisms. Results have shown that the Auction Share algorithm achieves high utilization of a proportional share scheduler and low latency of a Borrowed Virtual Time Scheduler.

P. Goldsack et. al. [25] describes Smartfrog (Smart Framework for Object Groups). Smartfrog aims to leverage the design, deployment and management of distributed component-based systems. It provides an infrastructure to standarize configuration and lifecycle management of applications. In addition, it takes care of fundamental issues such as scalability and reliability. Smartfrog implements a language [26] where components and systems are defined. This language describes software components that could be installed over hundreds of thousands of resources. The language supports configurations, components and systems. In addition, the configuration files could be validated prior to being executed over real scenarios. Descriptions in this language are converted in running distributed systems thanks to the underlying infrastructure of daemons running on whichever hosts are to be included in that

distributed system. Executed configurations are monitored by the Smartfrog infrastructure. Events happening while configurations are running could be mapped to triggers which generate actions on software or hardware components.

2.5 Summary

Grid computing has achieved important advances respect to definition and implementation of protocols and services to support the integration of scattered resources. Different grid implementations today can interoperate because there is a common ground of communication protocols along with resource management services (aka. GRAM and WS-GRAM) widely adopted in grid infrastructures. Different grid-based computational platforms have been deployed over local, regional and national networks that show significant computational achievements and offer an alternative computational infrastructure to drive solutions to challenging problems.

Grid aims to provide a general-purpose distributed computational infrastructure. This is a remarkable and challenging problem because grid-based computational models would consider different issues such as: application characteristics, workload characteristics, resource management policies, topological interconnectivity, and resource performance. At system level, different approaches have been made to model the behavior of static grid systems such as DAGs and Petri Nets. Since, static systems are unrealistic, most of grid dynamic models employ forecasting techniques based on data mining and different heuristics. At programming level, the problem has been directed to model grid assets as services in such a way that grid programs could defined in terms of the interaction of several services. Thus, workflow tools are to the grid systems as message passing tools are to the cluster systems. Resource management has explored different scenarios to range from application and resource deployment up to mapping tasks to resources. In particular, DLT proposes a simple solution to the scheduling problem of parameter sweep application over multi computer systems.

We have circumscribed our framework to undertake the problem of adaptive execution of parameter sweep applications over grid infrastructures. The adaptive problem has been tackled through statistical analysis. The statistical analysis considers previous performance exhibited by a particular application and predicts with a significat level of accuracy future

behaviour. The resource relevance is determined by user preferences such as performance, reliability and economic factors, among others. In addition, the load is distributed in a way that all the participating resources end at the same time. This framework is non-intrusive and relies heavily on trustable external information sources thus a pull-based mechanism is employed to get a partial snapshot of the system.

CHAPTER 3

A Computational Framework for Grid Environments

Grid infrastructures based on cluster legacy are characterized by heterogeneity and multiple administrative domains that provides an environment with high levels of uncertainty on availability and reliability. Meta-schedulers have been developed for orchestrating geographically spare resources. Often the proposed solutions do not consider realistic scenarios where grid infrastructures are interconnected through non-dedicated networks, and managed under diverse administration policies.

Falkon [27] is a framework that aims to leverage efficiently execution of many short tasks on large computer clusters. It combines multi-level scheduling, streamlined task dispatcher, anda data-aware scheduler. Falkon has shown excellent results running short tasks under Massive Parallel Processing (MPP) environments. It does not use any specific Local Resource Manager (LRM) for task dispatching but implements a customized module for task mapping into computational resources. Falkon is not intended for scheduling user's applications, but it executes efficiently scheduling plans coming from high level workflow tools such as Karajan. Gridway [28, 29] enables large-scale, reliable and efficient sharing of computing resources managed by different LRM within a single organization or scattered across several administrative domains. The architecture consists of four modules:request manager, dispatch manager, submission manager, and performance monitor. Every module is highly configurable through a scripting language designed for Gridway. Gridway provides a basic scheduling mechanism

based on flooding. During initial phases, remote resources are flooded with tasks in order to determine performance capabilities. Flooding is a greedy approach that takes a snapshot of the system, and consequently produces network congestion and resource overload.

The design and implementation of a grid computing enabling framework that provides functionalities for deployment, integration, and management of computational resources grid based is presented as follows.

3.1 Architecture

3.1.1 Grid Resource Architecture

Grid infrastructures are built from the integration of multiple computational resources which are managed by diverse administrative domains. Every grid resource is compounded by particular hardware and software elements. As consequence, grid systems are characterized by their heterogeneity since they embrace a plethora of hardware devices, operating systems, scheduler managers and monitoring tools, among other elements; many of them, non-interoperable at all. The way how users perceive the access to those resources is highly influenced by the management policies imposed by the resource managers.

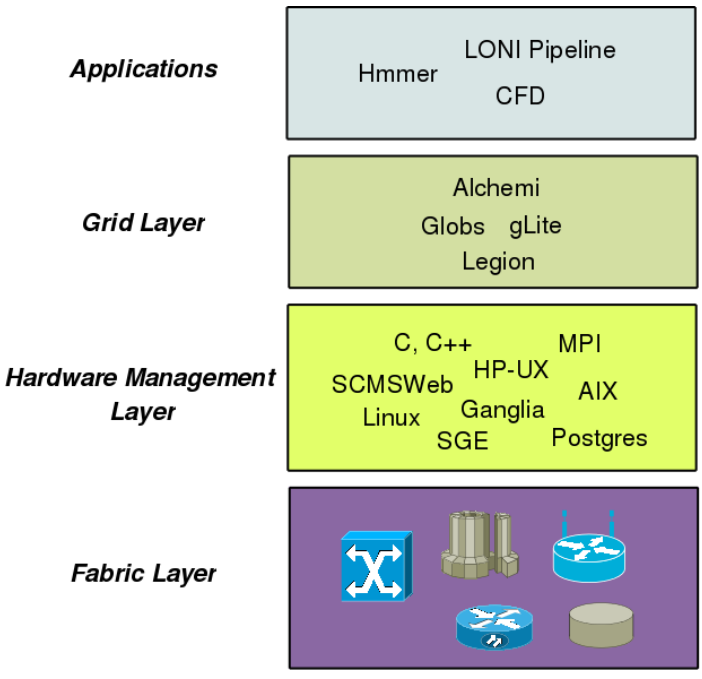

Figure 3.1. Grid Resource Architecture

Grid resources follow a multi-layer architecture which hides the specific details exposed by the hardware and software components. Each layer defines service access points which are used by the upper layer for accessing the methods and services offered by lower layers. Figure 3.1 depicts the hierarchichal structure observed by most of the grid resources. The fabric layer is located at the bottom of the hierarchy. It regards with physical components such as storage devices, processing units, and special measurement devices, among others. The Hardware Manager Layer (HML) at the next level, groups software and firmware components to hide the complexity and the heterogeneity found in the fabric layer. Software components such as operating systems, monitoring tools, development tools, system libraries, and Local Resource Managers (LRM) are part of this layer. This layer embraces those components which are managed by a single administrative domain. The following layer is known as the grid middleware. It provides a unique interface for accessing the functionality offered by the HML. Hence, it gathers common tasks in functional groups and provides single access points to specific operations regarding with HML components.

3.1.2 Gridjobs Architecture

Gridjobs is a computational platform for adaptive execution of sweep parameter applications. Gridjobs perceives the grid infrastructure as an observable system. It integrates different functional units in charge of collecting information provided by the environment and processing this information in a way that the assets behavior can be estimated in real-time. Figure 3.2 shows the modular design exhibited by the Gridjobs framework. This loosely coupled architecture allows that each module operates independently. A module in addition exposes its functionality through different communication protocols such as web services and RMI protocols. Next a description of the modules is given.

Persistence Module

The *Persistence Module* mimics an Information Service (IS) defined in conventional grid architectures. It is represented by entities that in most of the grid models are considered such as a user, task, and grid resource. However, the module is not compliant with any grid based IS, but exposes information through web service protocols, and makes information persistent over a

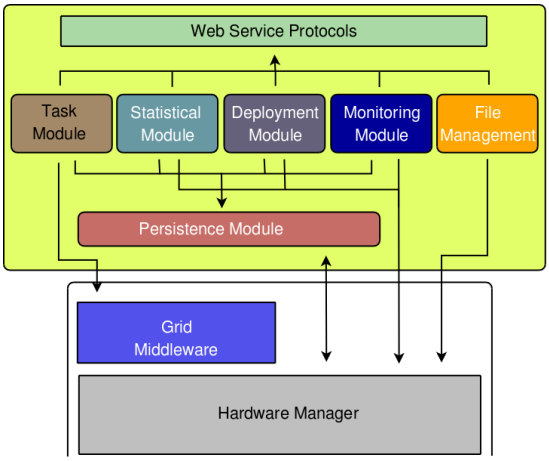

Figure 3.2. Gridjobs Modules

relational database instance. Different from other initiatives, our approach does not implement any particular query language or discovery strategy implemented in other implementations such as MDS and GRIS.

The Persistence Module relies on Grails object relational mapping implementation known as GORM [30] for storing the information associated with the entities defined in the grid model. (See appendix A.1).

The IS module is populated with information provided by a framework user and other framework modules while the framework runs. For instance, a user information is collected the first time when the framework is executed, the information associated to an application is gathered during the application deployment process and the grid resource and application performance are fed by the task monitor module.

Deployment Module

The *Deployment Module* provides a straight mechanism for deploying applications linux based over grid resources. Figure A.7 shows a screen shot of a client application used for deploying legacy applications over grid infrastructures. This client program collects information related to the application such as application name, executable name, remote installation path, remote work directory path, and the grid resources where the application would be deployed. Along with this information, the user must also provide the application source code and a basic script to contain the steps used for deploying the application in a conventional Unix system.

When this information is provided, the deployment process would be started by the user. The application source code, along with the script, is uploaded from the user computer to the framework server through the HTTP protocol. The framework then starts to transfer the script and application files to each selected grid resource through the SSH protocol. The module then executes the installation script on each resource and sends the information generated during the deployment process to the IS module.

Task Module

A task refers to executing a program in a cluster's compute node. The task module deals with executing and monitoring task. It supports asynchronous and synchronous task execution. Gridjobs is a highly parameterized platform. For instance, a configuration file specifies the monitoring frequency, the maximum number of computational nodes involved in an execution, per grid resource, and expectation times, among others. Hence, prior to executing a task, the user must indicate the expected task execution time(*application makespan*). This time information is used by the framework during task executions in order to avoid to indefinitely waiting for finalizing a task. The task module uses information stored in the IS in order to build commands for submitting task executions to grid resources via services provided by the Grid Resource Allocation Manager(GRAM). For instance, IS provides the LRM instance and name of the headnode, the task module contacts the local GRAM in order to request the task execution. Underneath GRAM establishes a secure and trustable connection with the remote grid resource which carries out the task execution.

Is already said. The framework invokes the `globus-job-run` for synchronous execution, whereas the `globus-job-submit` command is invoked when asynchronous execution is required. The later command is suitable for executing long lasting tasks. Since `globus-job-submit` does not wait for ending the submitted task execution, returns a *task identifier* which would be later used by the task monitor module in order to track task performance. Thus, the task monitor at regular intervals queries the task status through the `globus-job-status` command. Every succeeded task traverses four stages: UNSUBMITTED, PENDING, ACTIVE and DONE. Every time that some change occurs to the task's state, this change is recorded into the IS. The

cleaning and notification processes are carried out when the DONE stage is achieved. The `globus-job-clean` command is invoked to clean any orphaned processes or staged files created during the application execution. Finally, the user who submitted the task is notified via e-mail.

Although, the interaction with a remote LRM is achieved through the *globus-* series commands, this interaction is not hardcoded inside the framework but implemented in the *gridjobs.jar* library (Section 3.2). Thus, the `script` package defined in *gridjobs.jar* is modified for integrating the framework with other grid middleware implementations.

Monitoring Module

The *Monitoring Module* fulfills two requirements monitoring cluster status and application executions. The monitoring module relies on monitoring tools deployed at grid resource side for monitoring cluster. Those monitoring tools have be able to record events happening at compute nodes level. At this state, Gridjobs is able to interact with two of the most widely adopted monitoring tools in the cluster market, such as SCMSWeb and Ganglia. These tools are accessible via web services and TCP sockets, respectively. From the information that they provide, the monitoring module can infer resource capabilities, such as number of compute nodes, number of processors per node, memory size, and storage availability, among others. It can also process on-line as well as off-line data associated with the resources. For on-line or real time processing, Gridjobs uses two different communication protocols for interacting with the aforementioned tools. For SCMSWeb installments, Gridjobs uses web services (HTTP protocol and XML) as communication protocol. The framework then submits a HTTP GET request to a remote SCMSWeb instance and receives as a response a XML stream which contains the information associated with the computational nodes status. Gridjobs interfaces with SCMSWeb because is the monitoring tool suggested by Pacific Rim Appliation and Grid Middleware Assembly(PRAGMA). Ganglia uses in contrast raw sockets for showing resources status. Ganglia is another widely adopted monitoring tool and it is part of the Rocks distribution. Rocks counts with an active community of developers and it is considered as one of the most easy to install distribution for clusters. Gridjobs creates a socket connection with the remote resource and receives a XML stream to contain the computational nodes status.

Despite of both monitoring tools return a XML stream, each XML exhibits a particular structures, therefore, Gridjobs implementes two different parsers which are in charge of extract from the XML stream, relevant information to the local IS. The monitoring module processes log files retrieved from remote LRM for off-line analysis. At this time, Gridjobs only supports Sun Grid Engine(SGE) log files. Every SGE installment records the LRM activity in the *accounting* file. From data found in this file, it is possible to infer who the most active users are, what the characteristics of the most successful tasks are, application execution times and delay time of the users, among others. The off-line processing does not provide updated resource information, but does provide relevant historical data that is useful for supporting forecasting processes. On the other hand, the monitoring module also surveils application executions. Gridjobs is mainly concern to execute applications which load is arbitrarily divisible. Hence, when one execution is requested, the framework divides the application load amongst the available resources considering their historic performance. The way how the load is divided, highly depends on information provided by the statistical module, Section 3.1.2. The statistical module then infers the amount of load to each resource requires in such a way that most of the resources end their execution at the same time. Then, this module estimates the application termination time and schedules a thread which would care of all pieces achieve a termination state (DONE or FAILED). When all tasks finalize their executions, the thread finally reports the application execution time to IS.

Statistical Module

This module analyzes the behavior observed during task executions. It retrieves information from IS and estimates resource and task performance scenarios. Information associated with task executions is collected each time that an asynchronous execution is invoked. Each task monitor in charge of a particular task, records each event to happen during the task lifetime.

The statistical module processes the data collected and tests the observed values against several probabilistic functions and finds the function parameters to best model the data. This module employs the Kolmogorov-Smirnov test to determine the probabilistic function along with its corresponding parameters which exhibits the closest *p-value* to 0.5. KolmogorovS-

mirnov is used for quantifyng a distance between the empirical distribution function of the observed data and the cumulative distribution function of the reference distribution, or even between the empirical distribution functions of two samples.

Every time that this module is executed, Groovy and R [31] scripts to include recent systems activity are dynamically created. R is a statistical package which Gridjobs uses for carrying out the probability duties. This probabilistic approach allows to represent resource and task behavior with an important degree of accuracy. Finally, our framework has incorporated this probabilistic component in conventional models Divisible Load Theory (DLT) based in such a way that more precise scheduling plans could be generated. This approach has shown an improved representation of the grid resources because it considers their dynamic essence.

3.2 Implementation Issues

Gridjobs has been developed under Groovy [32] and Grails [33] platforms. Groovy is a programming language for the Java Virtual Machine. It supports all libraries developed for the Java language but also provides features inspired by other programming languages such as Python, Ruby and Smalltalk. Grails is a framework to support web applications development. Different from other web frameworks, it does not intend to re-invent the wheel but provides a modular architecture which allows the integration of technologies developed by third parties. Grails follows the Model-View-controller (MVC) paradigm. The Model represents application core entities. For instance, users, tasks and grid resources are entities defined in the model of the Gridjobs framework. These entities are made persistent through Hibernate technology. Hibernate interoperates with most popular databases in the market. Gridjobs in particular employs PostgreSQL as its backend. Controllers handle the application logic. In particular, Grails implements this concept through services and controllers per se.

Gridjobs in addition incorporates Quartz and Mail plug-ins in order to provide planning and notifications mechanisms to the framework components. Quartz is an open source job scheduling system developed in Java. It provides two fundamental classes, Job and Trigger. A Job class allows the definition of any task programmed in Java. However, a Job instance by itself is useless because it needs a Trigger to fire its execution. When a Gridjobs client requests

a task execution, a Trigger instance is scheduled for attending this request. When the trigger is fired, a `GlobusjobinstanceJob` instance is created. This instance submits the application execution to the grid and creates other Trigger that will fire a `GlobusjobstatusJob` instance, which in turn, surveils the application behavior. Under either normal or abnormal task finalization the `mail` plug-in is employed for creating an e-mail to contain a short report describing the termination status to the task owner.

The following sections present an algorithmic description of the task, deployment, statistical, and monitoring modules. According to how they are activated inside the framework, they are divided in two groups: synchronous and asynchronous modules. On one hand, task and deployment modules are asynchronous because they are activated by user or third parties requests through their web service interfaces. On the other hand, statistical and monitoring modules are synchronous because they operate on periodic intervals of time. However, additional wrappers have been also developed in order to allow asynchronous execution of the aforementioned modules.

3.2.1 Asynchronous Modules

There are two asynchronous modules, the task and deployment modules. Those modules are accessible from external entities through web service and RMI protocols. Appendix A.2 describes the implementation of these modules as Grails services and modules requirements for a proper interoperation between Gridjobs and grid resources are described as follow.

Deployment Module

This module provides a mechanism for easy deployment of legacy applications Linux based over computational grid resources (*aka.* clusters). This module relies on the SSH protocol for transferring the application files and executing the application installation scripts. Prior to deploy an application, it is necessary to provide to the Gridbjos framework textual information relevant to the application, the application files and installation script. The textual information describes insight details of the application such as executable name, application path, data path, among others. The application files expected by Gridjobs, are regular files used in conventional Unix systems for building legacy applications such as *makefile*, *configure* script

and source code, among others. In addition, this module allows to determine if a given application exists in a given grid resource and uninstall an application deployed via Gridjobs, see appendix A.3.1.

Task Module

This module provides execution and monitoring facilities over tasks submitted to Gridjobs framework. The framework decouples its dependency to any grid middleware implementation since it follows a layered approach where the technological details are handle in the *grid* package, implemented in the Gridjobs library.

The task module provides synchronous and asynchronous task execution. On one hand, under synchronous execution, the task owner is blocked until the task ends its execution. On the other hand, asynchronous execution does not block the caller but notifies her about the task finalization status employing the *mail* protocol. Gridjobs makes that functionality available through the `LaunchService` controller and accessible for third parties through the *Hessian* protocol.[1]

This module interacts with remote grid resources through the GRAM protocol and assumes a proper integration of GRAM with some Local Resource Manager (LRM) such as SGE, Torque or MAUI, among others. (Appendix A.3.2.)

3.2.2 Synchronous Modules

Synchronous modules are in charge to keep updated information associated with grid resources. The monitoring module interacts with remote monitoring tools and obtains the resource health. The statistical module periodically analyzes the information observed by Gridjobs and identifies probability functions to best model the observed resource behavior. Implementation details of these modules are provided in appendices A.3.3 and A.3.4, respectively.

3.2.3 Gridjobs Libraries

The Gridjobs framework relies in two libraries that we have developed in Groovy. First, *lib-gridjobs.jar* is a utility library to provide a single access point to third party technologies such

[1] Hessian is a binary web service protocol implementation.

as Hessian and Quartz. Basically, it pulls together code sentences into methods to reduce the algorithm length, leverage the coding speed, and hide the technology singularities. For instance, the `grid` package hides the technical details employed for interacting with a remote grid resource. However, if a user does not have access to the Globus Toolkit middleware but does have access to other grid implementations such as Unicore or G-Lite; the framework does not require any modification but the `grid` package does. Finally, *libgridjobs.jar* is also used by client modules because it supplies methods for managing graphical components, handling date/time data and remote communication via SSH and HTTP protocols, among others.

The second library is *libremote.jar*. It contains the interfaces which are implemented by the Gridjobs modules who expose its functionality through web service and RMI protocols. The framework as well the client applications need to access them. These interfaces represent, from the client point of view, the local reference to remote operations. For the Gridjobs modules perspective, they represent the contracts that modules must obey for exposing their functionality to remote entities.

3.3 Software Requirements

Gridjobs is a service-oriented framework implemented under Grails and Groovy programming environments. Gridjobs can be used as an application deployment tool and a gateway for adaptive execution of parameter sweep applications over computational grids. In addition, it integrates multiple services to watch and estimate the application performance over grid infrastructures. Therefore, Gridjobs integrates diverse external software projects for providing the functionality that an adaptive framework requires. These software dependencies are divided in application requirements and Grails requirements.

3.3.1 Application Requirements

Gridjobs allows the application execution over grid infrastructures. Thus, Gridjobs expects that the host where the framework is deployed along with the user who exectutes the framework exhibit the corresponding valid host and user certificates. Gridjobs also has a statistical module in charge of predict application performances. Using indirect observation, it records

the application execution, stores the observed information in a permanent repository and later forecasts the application performance. For storing the collected data, current Gridjobs release employs Gridjobs. However, any database supported by Hibernate [34] would does the job. For the statistical analysis, Gridjobs employs R [31]. Gridjobs interfaces with R through scripts generated dinamically by the framework statistical module. This module is periodically executed in order to keep an updated record of the observed application performance in such a way that the framework could generated adaptive application execution plans.

Application	Version	Homepage
Java SDK	1.6.0_11	http://java.sun.com/
Ant	1.6.5	http://ant.apache.org/
R	2.7.2	http://www.r-project.org/
PostgreSQL	8.3.1	http://www.postgresql.org/
Groovy	1.5.6	http://groovy.codehaus.org/
Grails	1.0.3	http://www.grails.org/

Table 3.1. Software packages required for deploying and executing Gridjobs.

3.3.2 Grails Requirements

Grails is a platform for developing web-based applications. It exhibits a modular design that allows to expand its functionality through plug-ins. Gridjobs relies on three plug-ins quartz, mail and remoting. Quartz is a java scheduler. It is used to execute the synchronous modules and the task monitor and application execution monitor. The mail plug-in is used to notify when tasks either finalize or abort their execution. Finally, the remoting plug-in provides the suitable environment to expose the service and module functionalities to third party entities through web services.

Plugin	Version	Homepage
Remoting	1.0	http://www.grails.org/plugin/remoting
Mail	0.5	http://www.grails.org/plugin/mail
Quartz	0.4.1-SNAPSHOT	http://www.grails.org/plugin/quartz

Table 3.2. Grails Plug-ins required for Gridjobs.

3.4 Summary

In this chapter, we have described the architectural design and implementation of a personal computing framework to launch application executions over operational grid infrastructures. Gridjobs interacts on top of computational grids to exhibit capabilities through grid services. Gridjobs follows a service-based loosely coupled architecture. In fact, Gridjobs is composed by different modules which interoperate with external entities through web service protocols. Gridjobs has divided the services between two groups: synchronous and asynchronous services. Statistical and Monitoring modules are synchronous. That is said because on periodic intervals of time the statistical analysis is launched in order to update the performance profile of the resources. Similarly, the monitoring module enquiries regularly remote monitoring tools in order to record resources status. However, these services show service access points to enable that users or other external components invoke their execution. Deployment, task and persistence modules are categorized as asynchronous modules. They are not executed on regular interval of times but on user demands. The deployment module has been used to install Linux-based legacy applications over clusters. This module assumes the availability of a protocol to support distributed data access (aka. Network File System (NFS) protocol) on the clusters. The task module allows the execution of legacy applications over computational grids. This module expects that each computational resource supports a grid-based resource management service which must be integrated with some LRM. The task module not only executes applications but also monitors them. It queries periodically to remote GRAM instances about the task status. Transitions observed during the task execution are reported to Gridjobs Information System.

Gridjobs has been developed under Groovy and Grails but it also interfaces with R for executing statistical routines to support the forecasting process. We envision a plug-in based platform which provide a more adaptive computational environment. For instance, Gridjobs supports different ranking schemes defined by the Gridjobs user through the configuration file(*application.properties*). This file defines a set of variables to drive the Gridjobs behavior.

CHAPTER 4

Resource Management

Grid computing defines Virtual Organizations (VOs) as abstract entities created on-the-fly from several resources governed by different administrative domains. Each VO orchestrates the different resources and presents a set of services to fulfill user demands. Constituent elements of a VO are not only circumscribed to processing units, storage devices or services but also others VOs.

Classical scheduling algorithms were designed with a steady computational platform in mind. Similarly, highly coupled parallel architectures present an operational environment characterized because of their high speed and low latency network links, low levels of communication failures between nodes, and well-known management policies, among others. Hence, these platforms present controllable characteristics that allow work to proceed in predictable environments. On the other hand, VO-based environments exhibit a different distributed computational environment with ubiquitious uncertainty conditions. Since VOs could be constituted by resources from different administrative domains whose performance is driven by particular management policies, the resource managers must incorporate scheduling algorithms to control this particular computational environment. This is a cumbersome task because VO addresses a wide range of application over an unfit infrastructure.

4.1 Constituent Grid Elements

Grid presents different infrastructure profiles because it intends to address solutions to diverse challenging problems. Thus, a Grid environment could be formed from compute nodes, clus-

ters, storage devices, simulators, and telescopes, among others. According to the services that each resource exposes, it would be categorized as a data grid, computational grid or service grid.

A data grid deploys specialized hardware and software components to provide reliable and efficient access to exceptional amounts of data. For instance, resources deploying grid-based libraries for storing medical images, distributed relational-object databases or computational resources to employ sophisticated array storage devices along with high performance filesystems to provide expeditious access to distributed files and objects.

A computational grid is characterized because of its astounding amount of computational nodes along with advanced hardware architectures to incorporate multi-core and hyper-threading processing units. Most of the members in a computational grid are called clusters. The clusters are accessible through Local Resource Managers (LRM) to implement basic scheduling algorithms driven by management policies defined by the cluster manager. In addition, compute nodes have homogeneous architecture and they are interconnected through reliable and low latency network infrastructure. Computational grids are also divided in high-throughput and distributed-supercomputing platforms. The former platforms increase the completion rate of a stream of jobs. The second kind of platform carries out parallel execution of applications employing multiple computational nodes mapped on different clusters.

A Grid service deploys dissimilar services to implement multiple modules and routines for different purposes. A Grid service leverages the creation of workflows to require streamlined service orchestration. The Grid service category can be further divided in on-demand, collaborative and multimedia grid platforms. Grid services are emerging as solid computational infrastructures because large corporations have pointed their resources toward service-oriented software architectures. This fact has leveraged the development of multiple workflows systems for bioinformatics, business process automation, business process management, business process modeling, and computer-supported collaboration, among others.

Our work focuses on computational grids. A description of the software components present in these Grids now follows.

4.1.1 Minimal software stack for a single computational grid resource

Each computational resource must deploy a middleware and a set of managers and services in order to be consider as a computational grid resource. The grid middleware would provide a set of services such as naming, monitoring, resource discovery, transfering, and job management. Those services present a common set of grid-compliant protocols and rules to leverage the interoperation and orchestration of multiple grid platforms.

For naming service and objects, there are different approaches such as relational, object models, language based and X.500/LDAP (Lightweight Directory Access Protocol), among others. On top of a naming scheme, the resource discovery service is implemented. In particular, Globus Toolkit has implemented two services for naming and discovering resources: Monitoring Discovering and System (MDS) and Grid Resource Information Service (GRIS). MDS4[35] implements a sophisticated scheme to support monitoring and discovery operations over dissimilar resources. It uses standard interfaces defined in WS-Resource Framework (WSRF) and WS-Notification (WS-N) specifications that embrace larger sets of resource data. Schopf et. al. [36] have shown that MDS4 exhibits better perfomance than other similar service directories such as MDS2, DataGrid Relational Grid Monitoring Architecture(R-GMA) and Hawkeye.

Protocols to support file transfer and staging mechanisms on loosely coupled infrastructures such as regional, national and international grids are essential. These protocols must expose high transfer rates, secure virtual channels and high levels of reliability even on public networks. Globus provides two protocols: GridFTP and Replica Location Service (RLS). GridFTP has shown important performance achievements over conventional file transfer protocols and services. Due to its versatile design, it is possible to develop enhanced transfer protocols and services using the initial GridFTP protocol [37]. In that sense, RLS has been developed on top of the GridFTP protocol. Among other characteristics, it provides mechanisms for achieving efficient data replication at suitable places in distributed grid environments, [38].

Monitoring tools are in charge of providing updated information from every compute node in a cluster. These monitoring tools interface with the node's operating system and retrieve

data such as load per node, amount of free, cached and used memory, network activity, free disk space, etc. These tools deploy agents on each monitored device which instead interact with local operating system services or through specialized monitoring protocols like SNMP. The information collected by each agent is sent to the central repository. This central component keeps historical records of each compute and summarizes and presents the cluster status in a structured way. For instance, Ganglia and SCMSWeb are two well-known monitoring tools. Ganglia is the monitoring tool that is installed by default in the Rocks distribution while SCMSWeb is the suggested monitoring tool for PRAGMA resources. Both use XML for structuring the cluster information, however they differ in their transport protocol. SCMSWeb uses HTTP and Ganglia employs plain sockets.

A service in charge of the job management in a computational grid is mandatory. Since computational grids are formed from clusters, it is relevant to know that each cluster deploys a particular LRM in charge of schedule and dispatch jobs. Request executions submitted to a particular LRM are highly affected by management policies and algorithms implemented in it. Globus implements the Grid Resource Allocation and Management (GRAM) service which provides a basic set of primitives for submitting, monitoring and cancelling jobs. This is a LRM-independent service which means that it could interoperate with any LRM deployed in the target computational resource. At this time, GRAM supports Sun Grid Engine (SGE), Portable Batch System (PBS) and Load Sharing Facility (LSF). Earlier versions of GRAM has evolved toward more standardized communication protocols in such a way that GRAM service could not be only accessed through command line tools and APIs but also through web service protocols. It is remarkable that GRAM does not provide any scheduling mechanism but it delegates this functionality to the underneath LRM.

4.2 PRAGMA - A Computational Grid Testbed

PRAGMA [39] is a grid platform to integrate different computational resources scattered around the world. PRAGMA is an initiative founded in 2002. Today PRAGMA counts with 25 members from 18 countries. Each member shares at least one cluster. Thus, PRAGMA is a computational grid constituted by 30 clusters, 458 compute nodes, 1020 CPUs, over 1.3

TB in Random Memory Access (RAM) and approximately 25 TB in disk space. PRAGMA is characterized because its hardware and software heterogeneity. All sites support GRAM and some of them Web Services GRAM (WS-GRAM) too. Most of the clusters are built from i686 compute nodes but there are also x86_64, Itanium and Power5+ systems-based. Similarly, there are diverse technologies for clusters scheduling, such as SGE(16), Torque(8), PBS(3), LSF(1) and LoadLeveler(1). Hence, PRAGMA could be denominated as a heterogeneous computational grid.

PRAGMA presents a flat management structure. There is not central authority to determine global management policies so that local policies established by each resource owner to drive the way the resources are used. There is not one unique central certificate authority (CA) in charge to emit certificates. Thus, any authorized CA could discretionally emit host and user certificates in such a way that its authorized certificates are automatically accepted by any other PRAGMA member. Similarly, any resource and user with updated certificates could dispatch tasks to any other PRAGMA resource. In that sense, PRAGMA could be extrapolated as a peer-to-peer computational grid.

Now, the decentralized nature of PRAGMA resources presents a tremendous challenge for determining the performance that each cluster could exhibit. Different techniques to estimate resources performance have been probed such as heuristics, pricing models, machine learning, and probability distributions. However, any estimation mechanism per se is useless unless that it comes companied by a feedback procedure to adapt and reconfigure execution plans according to the observed executions.

4.3 Scheduling Taxonomy

Because of the amount of components and problems intended to be tackled, Grid could barely be represented by one single architectural model. Thus, it is possible to identify diverse computational models applied to Grid infrastructures. Some works direct their attention at the way the different computational elements are interconnected or the management hierarchy that they could exhibit. That kind of model can be called topology-based Grid models. Despite previous works consider a wide range of network topologies from well-known topologies such

as bus, tree and rings; up to more sophisticated such as torus and hypercubes; many of them hardly represent real Grid infrastructures. For instance, Grid initiatives such as TeraGrid and PlanetLab exhibit a centralized management architecture where the deployment and execution of applications are carried out by a central manager. In contrast, PRAGMA presents a totally decentralized infrastructure where each user is responsible for deploying and scheduling the execution of the user's applications.

Scheduling is a widely studied problem in different areas where efficient resource management is a fundamental issue such as control theory, operations research and production management. Cassavant and Kuhl [40] present an interesting taxonomy applicable to conven-

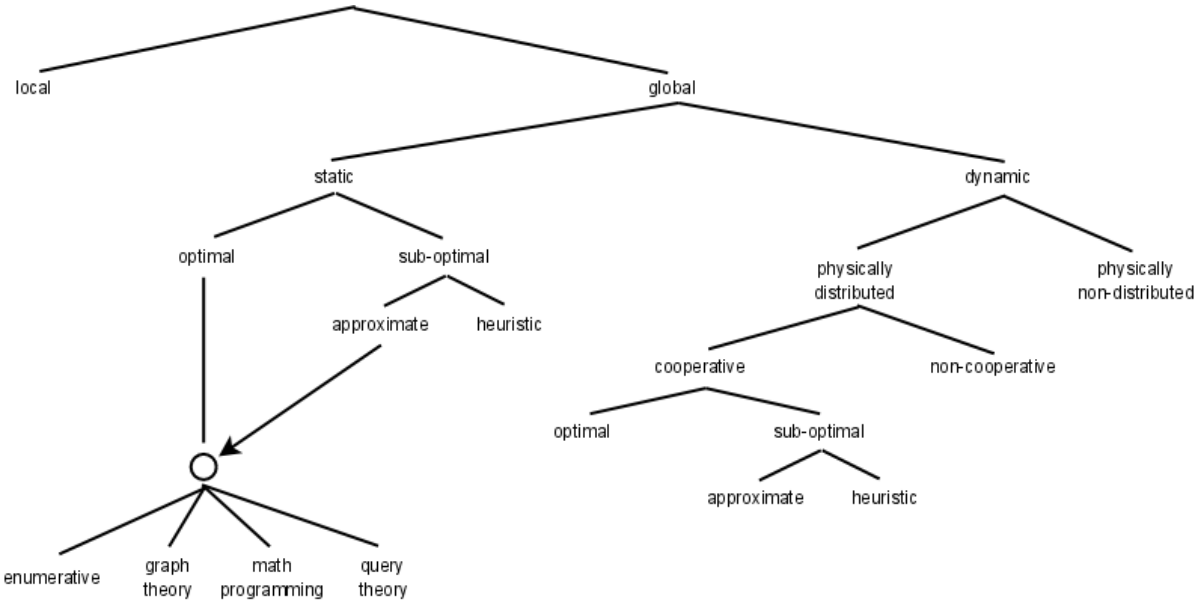

Figure 4.1. Scheduling Taxonomy proposed by Cassavant and Kuhl.

tional distributed systems, Figure 4.1. They conceptualize the scheduling problem as a problem with three actuators: consumers, resources and schedulers. In this context, the scheduler element deploys different policies to drive the generation of execution plans. However, utility computing platforms impose additional factors in the scheduling equation not being considered in that study. Thus, consumers present their computational and economical expectations represented as QoS (Quality of Services) and resource owners display their utilization policies and costs. Now, schedulers operating on these environments are required to consider the

constraints imposed by consumers and resources and generate execution plans to stick to the aforementioned constraints, Figure 4.2.

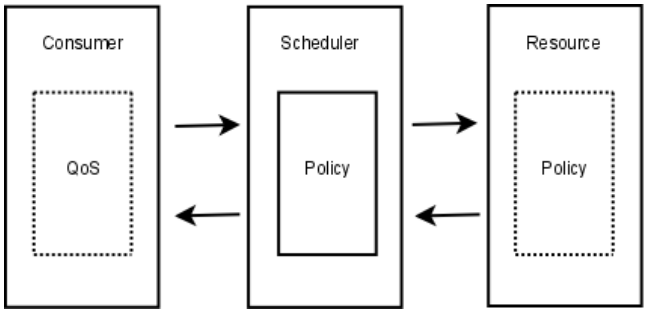

Figure 4.2. Current large scale infrastructures expose resource and consumer constraints to affect the execution plans generated by schedulers.

Rotithor [41] proposes another taxonomy for distributed systems but this time he considers dynamic distributed systems. In this context, the scheduling problem grows in complexity because not all variables used traditionally to generate execution plans are known. Hence, the scheduling problem is divided in two sub-problems: state estimation and decision making. On computational environments such as computational grids, where resources are governed by different administrative domains and the computational resources are interconnected via public networks, the problem of resource estimation is challenging. Rotithor classifies estimation approaches around to the following questions:

1. Who is in charge of collecting the resources information?
2. During state information exchange, how many elements are involved?
3. How the local information is disseminated?
4. How frequent the information exchange occurrs?

To the first question, centralized, decentralized and hybrid approaches are possible. In a centralized scenario, all the information is found in a unique repository. This kind of approach is present in cluster systems. The decentralized approach presents a scenario where there is not a unique information repository but many. Peer-to-Peer (P2P) based systems work under this approach. The hybrid approach, is present in computational grids. For instance, a computational VO monitors the status of the resources that form it and each resource instead

implements a centralized monitoring tool.

The second question refers to the consensus problem. Similarly, three possible responses are possible: complete, partial and variable. Partial and variable are approaches related closely but in a partial scenario a minimum number of nodes is always required, for instance the half plus one. In a variable scenario an arbitrary number of nodes is enough.

The third question tackles the problem of deciding who initiates the transmission of a resource status. and three possibilites are considered: voluntary, involuntary, or combined. In the first scenario, each element that identifies changes in the environment, immediately would inform whom has been registered for notification. Involuntary dissemination occurs when an information provider is directly asked for the information.

The four questions concern the frequency of information interchange. The author visualizes three possibilities: periodic, aperiodic or combination. In a periodic frequency, the information is exchanged on regular time interval. Aperiodic interchange is triggered when certain conditions are met and the combined solution is relevant on computational environments that handle real-time problems as well as general purpose computations.

4.4 Gridjobs Resource Management

4.4.1 Scheduling Approach

Gridjobs is a metascheduler to orchestrate several Grid-enabled resources for executing parameter sweep applications. Following the approach suggested by Rotithor, Gridjobs has divided this orchestration process in two sub-problems. The first sub-problem is resource performance estimation and the second one is task distribution.

Resource Performance Estimation

Gridjobs is an application-aware meta-scheduler. It assumes that the resource performance is highly determined by the application that this resource runs. That is said because two different applications executed with inputs of similar sizes could exhibit different execution times. Thus, Gridjobs sends dummy executions to the resource under observation and watches how the application instance performs in the given resource.

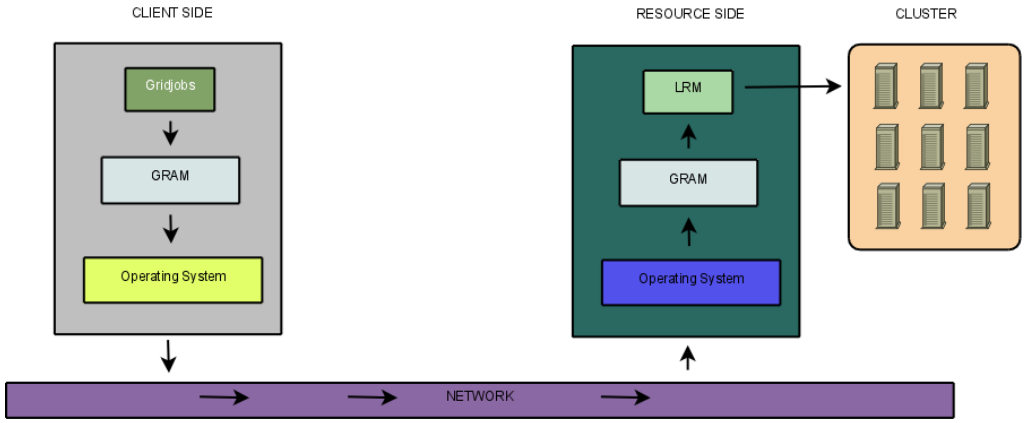

Figure 4.3. Interaction perceived by Gridjobs.

Gridjobs runs on user-space and it interacts with a local GRAM service instance, Figure 4.3. When Gridjobs submits an execution, the local GRAM returns an identifier associated with the submitted task. Onward Gridjobs uses this identifier to track the application performance. Gridjobs first determines the time that this execution lasts in UNSUBMITTED state. This state occurs while the execution request remains in the local GRAM but it has not been dispatched and received on the remote GRAM. When the remote GRAM acknowledges the requested execution, this execution passes to the remote LRM. Then, the remote LRM enqueues the request. At this time, Gridjobs starts to measure the time that this execution lasts enqueued. This state is known as PENDING. Later, when the remote LRM finds a free slot where the execution can be executed, it then maps this execution to this free slot. Now, Gridjobs measures the execution time which is known as the ACTIVE time. For non MPI applications, the UNSUBMITTED and PENDING states are independent of the parameters passed to the application. However, the ACTIVE time is directly affected by the execution parameters. Noted that for MPI applications which specify the number of slots required for carrying out the execution, the PENDING time varies depending of the LRM management policies.

Along with the resource performance estimation, Gridjobs also cares for the resources state. Gridjobs then interfaces with Ganglia and SCMSWeb monitoring tools, Figure 4.4. Those monitoring tools, via XML, provide a complete information about cluster status such as number of compute nodes, number of dead nodes, free memory per node, free disk space

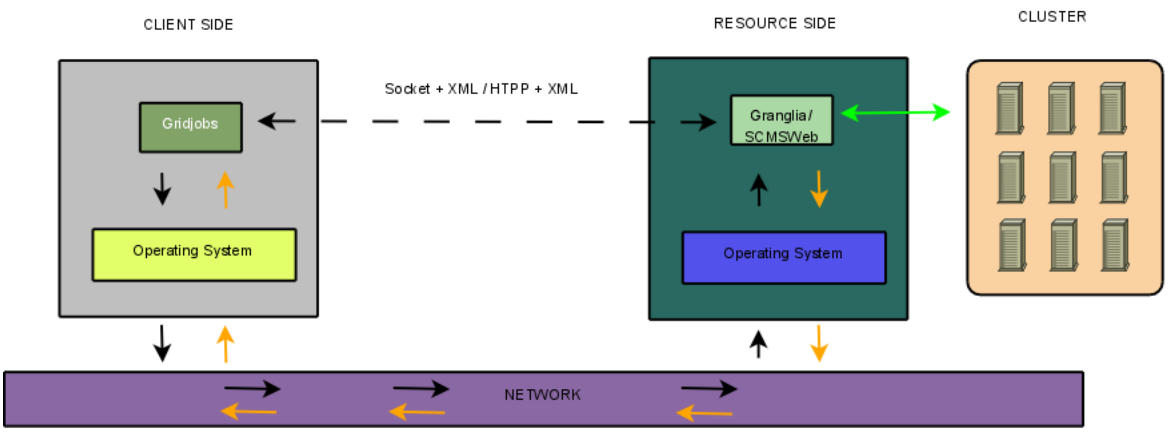

Figure 4.4. High-level description of interaction between Gridjobs and Monitoring Tools.

per node, and load per node, among others. Gridjobs parses this information and stores it in a permanent repository. Gridjobs does not require the global status of the computational grid to be known but only partial information, in particular of resources involved in the execution. Gridjobs employs periodic pull-based mechanisms that do not hinder with the normal monitoring tool operation at cluster level. In particular, Gridjobs is interested in the number of nodes per resource, how many nodes are alive and the *load15* value.

In Unix systems, *load15* indicates the average system load over the last 15 minutes. When Gridjobs enquiries some cluster monitoring tool, it receives the cluster status information and averages the *load15* measurement from all compute nodes. Finally, Gridjobs stores the averaged *load15* value in the Gridjobs Information System (GIS).

Task Distribution

For task distribution, Gridjobs envisions PRAGMA as a non-hierarchichal set of computational assets. Thus, Gridjobs establishes a star topology, Figure 4.5. Gridjobs can be seen as a gateway able to execute applications over different grid computational assets. The center of the star corresponds to the resource to host Gridjobs and the leaf nodes correspond to computaional resources. The server where Gridjobs is hosted could act as source and sink resource willing not only to submit executions but also to participate in application executions. Figure 4.6 shows a graph-based pictorial representation of a Gridjobs-based computational grid.

Gridjobs acts as the center of the star and the extremes of the edges represent grid

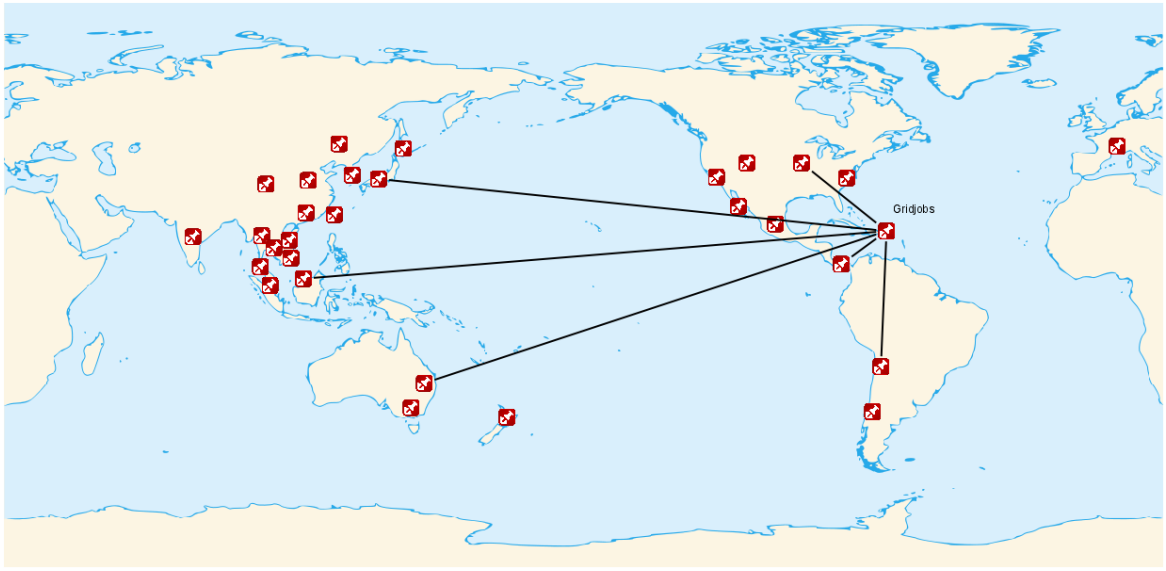

Figure 4.5. Gridjobs exhibits a star network topology.

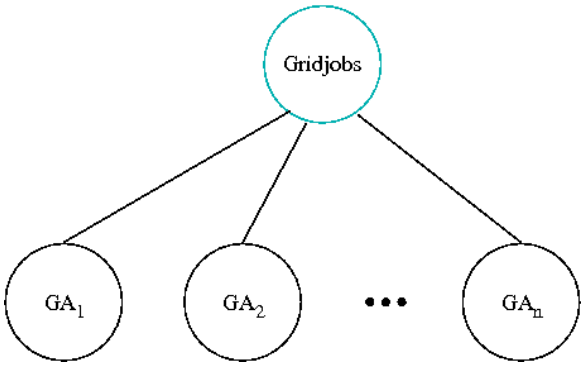

Figure 4.6. Gridjobs-based computational grid

computational assets (GA). In this case, those GA are clusters. Each GA deploys a particular LRM which can seen as a broker for the computational nodes under this cluster. Hence, Gridjobs is aware of the number of computational nodes per GA but it can not send submissions to particular nodes. Figure 4.7 explodes each GA and reveals the constituent macro-elements to define a GA. Thus, each GA deploys a Local Resource Manager (LRM) such as Condor, Portable Batch System, Sun Grid Engine, etc. Most of these LRM-controlled clusters are formed by homogeneous resources. This fact presents a great advantage because although Gridjobs can not submit directed executions to any particular compute node, that does not matter because all nodes have equal performance characteristics and it is assumed that efficient load balancing algorithms along with fair management policies are deployed at each LRM.

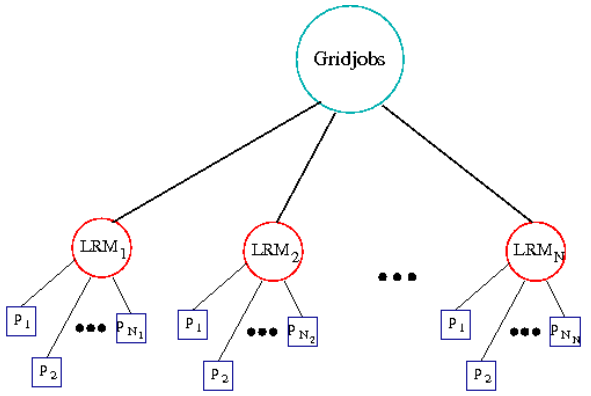

Figure 4.7. The Extended Logic Star Topology

A notation to describe those computational elements follows. Thus, let $\mathfrak{G}$ be a computational grid resulting in the integration of multiple computational resources which are scattered geographically and managed by diverse administrative domains. Hence, it can be said that $\mathfrak{G} = \bigcup_{j=1}^{m} g_j$, where m is the total number of computational resources. Now, each g_j can be seen as a tuple $\{R_j, P_j\}$, where R_j represents physical resources and P_j represents the policies and factors aside to drive the performance and behavior of the resources deployed in g_j.

Depending of the grid profile of R_j, it could represent different kinds of resources. For instance, for a data grid, R_j would represent storage devices and databases. In our case, R_j represents a set of homogeneous computational devices. Thus, $R_j = \bigcup_{k=1}^{m_j} r_{jk}$, where each r_{jk}

represents a computational node. Now, since P_j represents rules and constraints to affect the resource performance, Gridjobs visualizes P_j as a black-box, it then maps P_j and R_j as a probability function.

In grid environments there are an important list of factors that could affect the perceived resource performance, for instance overhead imposed by grid middlewares, irregular communication latency present over public networks, outdated management tools, uncertain resource load, unknown algorithms and management policies that drive the behavior of LRMs, and so forth. In particular, during our experiments we found that some resources give higher priority to tasks that require one computational node to run than those tasks that require two or more computational slots such as MPI-based tasks. In addition, many resources present integration problems and others have outdated management scripts.

Since Gridjobs works with parameter sweep applications, now we presented how the tasks are distributed amongst the GA. Let X be the input of a particular application app_1. app_1 does not require any kind of interaction with external entities in order to process its input data and X can be arbitrarily divisible in $x_1, x_2, \ldots, x_n$, such that

$$\sum_{i=1}^{\mathfrak{N}} x_i = X. \tag{4.1}$$

Now, Gridjobs divides X amongst the available resources in such a way that all the resources participating in the execution, end at the same time, Figure 4.8.

According to Figure 4.8, the time estimated for executing an application with input X is T. T is determined by the characteristics of the p_1 along with the overhead imposed by the different active and passive factors intervening the execution in p_1. Now, we claim that:

$$T = \mu_1 + \rho_1 + \frac{x_1}{\omega_1} \tag{4.2}$$

where μ_1, ρ_1 and $\frac{x_1}{\omega_1}$ (x_1 input size and ω_1 power processing) represent the elapsed times in UNSUBMITTED, PENDING and ACTIVE stages, respectively. These values are not constant but uncertain and they are affected by different elements. Figure 4.9 provides a pictorial

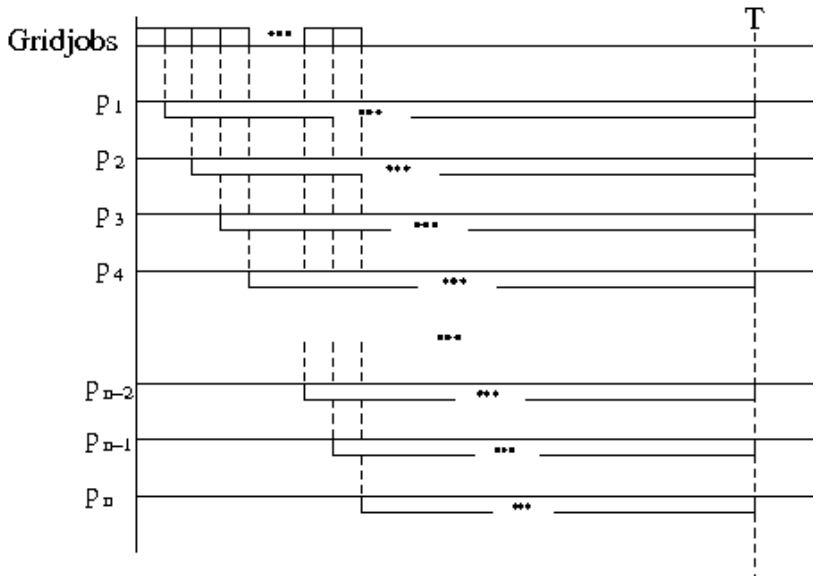

Figure 4.8. A Control Node sends job submissions to each node according to the schedule.

description of the elements affecting the execution of a given application over a grid infrastructure. When Gridjobs sends a execution request, the local GRAM addresses it. The local GRAM returns a request identifier to Gridjobs and tries to deliver the request to the remote GRAM. This delivery process could be affected by network congestion and load presented in the server to host the remote GRAM. When the remote GRAM receives the request, it then delivers the request to LRM. The total execution time is now affected by the number of pending tasks in LRM besides of the management policies used to determine the order and priority of the pending executions. Finally, when LRM maps the request to some *free* slot is when the effective execution occurs.

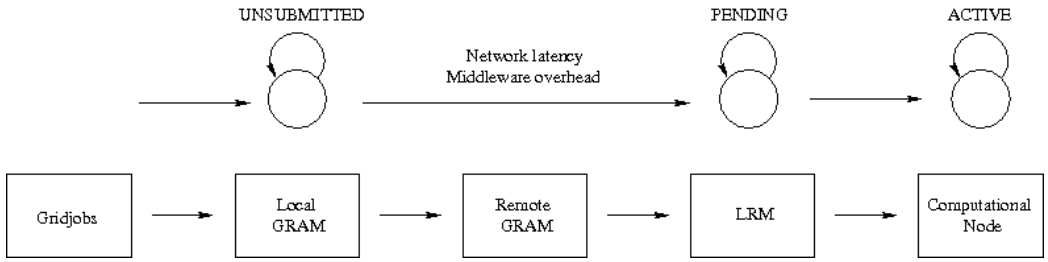

Figure 4.9. A diagram representing the different software components involved in a grid execution along with a diagram state.

Thus, to express the T value as a sum of constant terms is an unrealistic approach.

Equation 4.2 is then re-written as follows:

$$T = \mu_1'(\xi_\mu) + \rho_1'(\xi_\rho) + \frac{x_1}{\omega_1'(\xi_\omega)} \tag{4.3}$$

where $\mu_1'(\xi_\mu)$, $\rho_1'(\xi_\rho)$ and $\omega_1'(\xi_\omega)$ are probabilistic functions to model the behavior of UN-SUBMITTED, PENDING and ACTIVE stages, respectively. (ξ_μ, ξ_ρ and ξ_ω are the parameters of the aforementioned probabilistic functions.) However, for simplicity, subsequently equations stick with the nomenclature of Equation 4.2.

Now, the computational time of the second node is given by:

$$\mu_2 + \rho_2 + \frac{x_2}{\omega_2} \tag{4.4}$$

The second execution request is sent Δ time units later than the first request, Figure 4.8. So, the second request lasts:

$$T = \Delta + \mu_2 + \rho_2 + \frac{x_2}{\omega_2} \tag{4.5}$$

Now, from 4.2 and 4.5.

$$\mu_1 + \rho_1 + \frac{x_1}{\omega_1} = \Delta + \mu_2 + \rho_2 + \frac{x_2}{\omega_2} \tag{4.6}$$

x_2 is expressed in terms of x_1:

$$x_2 = \left((\mu_1 - \mu_2) + (\rho_1 - \rho_2) + \frac{x_1}{\omega_1} - \Delta \right) \omega_2 \tag{4.7}$$

Equation 4.7 is re-writen as:

$$x_2 = A_1 + B_1 x_1 \tag{4.8}$$

where, $A_1 = (\mu_1 + \rho_1 - (\mu_2 + \rho_2 + \Delta))\omega_2$ and $B_1 = \frac{\omega_2}{\omega_1}$. Since x_2 represents a chunk of data it must be positive, then A_1 and B_1 must be positive, too. According to that, $A1 \geq 0$ so $\mu_1 + \rho_1 \geq \mu_2 + \rho_2 + \Delta$ must be true. That inequality suggests a particular order for the processing nodes participating in the computation.

Considering again the expression 4.8, it can be said that $x_{i+1} = A_i + B_i x_i$. To find an expression to represent x_i in terms of x_1, lets see how x_4 can be written in terms of x_1. By

definition

$$x_4 = A_3 + B_3 x_3 \tag{4.9}$$

Now, let us replace x_3 in terms of x_2

$$x_4 = A_3 + B_3(A2 + B_2 x_2)$$

$$x_4 = A_3 + B_3 A_2 + B_2 B_3 x_2$$

By replacing x_2, we have

$$x_4 = A_3 + B_3 A_2 + B_2 B_3 (A_1 + B_1 x_1)$$

$$x_4 = A_3 + B_3 A_2 + B_2 B_3 A_1 + B_1 B_2 B_3 x_1$$

Now, it is easy to infer a general equation for x_i

$$x_i = \sum_{j=1}^{i-1} A_j \prod_{k=j+1}^{i-1} B_k + x_1 \prod_{j=1}^{i-1} B_j \tag{4.10}$$

Equation 4.10 gives us a closed-form expression for x_i. Now, we look for the value of x_1. Replacing x_i in Equation 4.1.

$$\begin{aligned}
&\sum_{i=1}^{\mathfrak{N}} x_i = X \\
&\sum_{i=1}^{\mathfrak{N}} \left(\sum_{j=1}^{i-1} A_i \prod_{k=j+1}^{i-1} B_k + x_1 \prod_{j=1}^{i-1} B_j \right) = X \\
&\sum_{i=1}^{\mathfrak{N}} \sum_{j=1}^{i-1} A_i \prod_{k=j+1}^{i-1} B_k + \sum_{i=1}^{\mathfrak{N}} x_1 \prod_{j=1}^{i-1} B_j = X
\end{aligned} \tag{4.11}$$

And now it is easy to derive x_1:

$$x_1 = \frac{X - \sum_{i=1}^{\mathfrak{N}} \sum_{j=1}^{i-1} A_i \prod_{k=j+1}^{i-1} B_k}{\sum_{i=1}^{\mathfrak{N}} \prod_{j=1}^{i-1} B_j} \tag{4.12}$$

4.4.2 Resource management algorithms

In this section, three algorithms are presented. The first algorithm is employed to find the different probability functions to model a resource behavior considering the application under execution. The second algorithm determines the total nodes available per resource and divides the load among the computational resources. The third algorithm implements different resource ranking approaches.

```
startanalysis()
 1  probfunclist ← [cauchy, exponential, gamma, geometric, logistic, lognormal, normal, poisson, weibull]
 2  for each resource r in DB
 3       do unsubmittedlist ← []
 4          pendinglist ← []
 5          activelist ← []
 6          for each task t in DB where t.resource = r
 7               do if t.exitstatus = DONE
 8                    then unsubmittedlist ← concat(unsubmittedlist, t.pending − t.unsubmitted)
 9                         pendinglist ← concat(pendinglist, t.active − t.pending)
10                         activelist ← concat(activelist, t.done − t.active)
11                    else unsubmittedlist ← concat(unsubmittedlist, −∞)
12                         pendinglist ← concat(pendinglist, −∞)
13                         activelist ← concat(activelist, −∞)
14          replaceMinusInfMax(unsubmittedlist, max(unsubmittedlist))
15          replaceMinusInfMax(pendinglist, max(pendinglist))
16          replaceMinusInfMax(activelist, max(activelist))
17          for each probability function pf in probfunclist
18               do generateRscript(unsubmittedlist, pf, r)
19                  generateRscript(pendinglist, pf, r)
20                  generateRscript(activelist, pf, r)
21  for each resource r in DB
22       do for each stage st in [unsubmitted, pending, active]
23               do [pf, params] ← selectProbFunctionWithMinp-Value(st, r, probfunclist)
24                  createRandomGeneratorScript(pf, parms, st, r)
```

Statistical Analysis

Gridjobs uses probabilistic functions to estimate the application behavior in a determined resource. Gridjobs divides the application execution time in three stages UNSUBMITTED, PENDING and ACTIVE. It is also aware that the exhibited performance in any stage is unrelated to any other stage. Finally, Gridjobs also considers that the application execution time is highly affected by the characteristics of the resource where the application runs.

The *startanalysis* algorithm is presented in Figure 4.4.2. This algorithm analyzes the performance of each resource known by Gridjobs, Line 2. Each resource performance is derived from information stored in the `task` table. This table stores different timestamps to record the changes of state that every task experiments during its execution, Lines 6-13. Thus, there are timestamps for the next transitions: UNSUBMITTED (`task.unsubmitted`) $\rightarrow$ PENDING (`task.pending`), PENDING $\rightarrow$ ACTIVE(`task.active`) and ACTIVE $\rightarrow$ DONE(`task.done`). Failed executions are specially treated because some timestamps could have invalid values, Line 11-13. In that case, the statistical module penalizes these executions assigning the highest timestamp observed for that resource, Line 14-16. When the information associated with each resource is gathered, R scripts for each stage and each resource are created, Line 17-20. Finally, the different probability functions are evaluated through the Kolmogorov-Smirnov test. The probability function to exhibit the closest value to 0.5 is then selected as the probability function to model a determined resource and state.

Numerical Example This section presents a numerical example to find a probability function for the ACTIVE state. The data below were observed in *komolongma* on June 1, 2009.

```
239980.5
239997.25
239998.5
239998.75
240000.5
239998
239998.75
```

Probability Function	Parameter Values
Cauchy	location = 239998.60 scale = 1.06
Exponential	rate = 4.05
Geometric	prob = 4.05×10^{-6}
Logistic	location = 239998.81 scale = 7679.97
Log-Normal	meanlog = 12.41 sdlog = 0.07
Normal	mean = 246661.25 sd = 18850.31
Poisson	lambda = 246661.25
Weibull	shape = 10.06 scale = 254896.53

Table 4.1. Parameters of several probability functions found during the performance evaluation of *komolongma*

240001

299978

From these data, the statistical module creates R scripts for the following probability functions: Cauchy, Exponential, Gamma, Geometric, Logistic, Log-Normal, Normal, Poisson and Weibull. Figure 4.10 shows a R script to find the parameters of the Cauchy probability function (`location = 239998.602803826` and `scale = 1.06432187998584`). Similarly, scripts

```
#!/home/jas/root/bin/Rscript
library(MASS)
v <- c(239980.5,239997.25,239998.5,239998.75,240000.5,239998,239998.75,240001,299978)
x <- fitdistr(v,"cauchy")
output <- paste(x$estimate[["location"]],x$estimate[["scale"]],sep="|")
print(output)
```

Figure 4.10. R script generated by the Gridjobs statistical module.

for the other probability functions are also created. Table 4.1 presents the parameters found for the other functions. Finally, the Kolmogorov-Smirnov test is executed over all the probability functions. Figure 4.11 presents a R script to execute the aforementioned test for the Cauchy probability function. In that case, Cauchy was the selected function.

Table 4.2 shows the p-values found.

```
#!/home/jas/root/bin/Rscript
library(fBasics)
v <- c(
239980.5,239997.25,239998.5,239998.75,240000.5,239998,239998.75,240001,240001)
ks.test(v,"pcauchy",location=239998.60,scale=1.06)
```

Figure 4.11. R script to apply the Kolmogorov-Smirnov test over a Cauchy probability function.

Probability Function	**p-value**
Cauchy	0.9559
Exponential	0.00189
Geometric	0.00189
Logistic	0.02225
Log-Normal	0.001313
Normal	0.001313
Poisson	3.04×10^{-8}
Weibull	0.004741

Table 4.2. Parameters of several probability functions found during the performance evaluation of *komolongma*

Scheduler

Figure 4.12 presents a pseudo-code description of the scheduler module implemented in Gridjobs. Given an application (`appname`) and range of data (`lowvalue` and `highvalue`), the scheduler divides the load amongst the computational resources where the required applications has been deployed. This module first determines the set of resources where the application has been installed, Line 1. If none resource is found, the execution ceases. Otherwise, this module estimates the performance of each resource, Line 6-11. This estimation is used later to determine the amount of data that each resource is able to process given a deadline. Then, a ranking scheme is employed to specify an order in which the load would be distributed amongst the resources, Lines 12-13. This ranking scheme is not fixed but determined in execution time by the user. Noted that different ranking schemes have been deployed on Gridjobs(Section 5.5.4) but it is possible to extend the base of available ranking schemes. The scheduler determines the number of total nodes available per cluster and finds the load size per each computed node, Lines 14-19. The load size is determined according to the equations 4.10 and 4.12. Gridjobs implements a function to estimate the load size and the pseudo-code is presented in Figure 4.13. Finally, the scheduler starts to submit the executions

```
scheduler(lowvalue, highvalue, stride, appname)
    ▷ Determine what resources have installed the 'appname' application
 1  resourcelist ← selectResources(appname)
 2  if size(resourcelist) <= 0
 3     then ▷ No available resources, then EXIT
 4
 5  estimationlist ← []
    ▷ Loop that estimates the time that a node in r
    ▷ takes executing 'appname'
 6  for each resource r in resourcelist
 7        do ▷ The R script estimates how long time some task spends
              ▷ in UNSUBMITTED stage in resource r
 8            unsubmittedtime ← executeRScript(UNSUBMITTED, r)
              ▷ Estimating how long time a task spends in PENDING stage at r
 9            pendingtime ← executeRScript(PENDING, r)
              ▷ Estimating how long time a task spends in ACTIVE stage at r
              ▷ considering the appname application
10            activetime ← executeRScript(ACTIVE, appname, r)
11            estimationlist[r] ← unsubmittedtime + pendingtime + activetime
12 resourcelist ← rankedList(estimationlist, appname, rank)
13 sort(resourcelist)
14 totalnodes ← 0
15 for each resource r in resourcelist
16       do totalnodes ← totalnodes +availablenodes(r)
17 numchunkspernode ← []
18 for i ← 1 to totalnodes
19       do numchunkspernode[i] ← selectLoad(i)
20 base ← lowvalue
21 for i ← 1 to totalnodes
22       do resource ← determineResource(resourcelist, i)
23          submitTask(base, base + numchunkspernode[i], resource, appname)
24          base ← base + numchunkspernode[i]
25          if base > highvalue
26             then break
```

Figure 4.12. Pseudo-code of the scheduler module deployed in Gridjobs.

to the selected nodes with their corresponding job sizes.

selectLoad(i)

1 **if** $i = 1$

2 **then return** $\dfrac{x - \sum_{i=1}^{\mathfrak{N}} \sum_{j=1}^{i-1} A_i \prod_{k=j+1}^{i-1} B_k}{\sum_{i=1}^{\mathfrak{N}} \prod_{j=1}^{i-1} B_j}$

3 **else return** $\sum_{j=1}^{i-1} A_j \prod_{k=j+1}^{i-1} B_k + x_1 \prod_{j=1}^{i-1} B_j$

Figure 4.13. Pseudo-code of the *selectLoad* function.

Ranking Schemes

Gridjobs has implemented four different ranking schemes whose pseudo-code is presented in Figure 4.14. At this time, these ranking schemes only consider performances exhibited by grid resources. For instance, the *timeexecutionerror* scheme ranks each resource according to the precision estimation. The *expansionfactor* evaluates the the application execution time along with the enqueued elapsed time. By the way, this ranking scheme is useful on scenarios where MPI-based applications are required. Some LRMs deploy management policies to establish the application priority according to the number of computational nodes that the application requires. Thus, executions to request one compute node go first. The priority then decreases when the number of compute nodes required, increases. Finally, Gridjobs has also implemented a ranking scheme to evaluate the computational nodes load (Section 5.5.4 provides more description about the aforementioned ranking schemes). However, other ranking schemes to regard with economic models such as bids and auctions, could be also implemented.

4.5 Summary

Grid infrastructures can be categorized in diverse and orthogonal taxonomies. According to capabilities exhibited by a grid resource, it could be classified as computational grid, data grid and service-based grid. Other taxonomy could be derived according to the management scheme to rule the resources utilization in P2P grids and centralized grids. PRAGMA is a

```
rankedList(estimationlist, appname, rank)
 1  if rank = DEFAULTRANK
 2     then return estimationlist
    ▷ 'estimationlist' is a hash map whose 'keys' are the resources name
 3  resources ← keys(estimationlist)
 4  resourcelist ← []
 5  for each resource r in resources
 6        do
 7           acum ← 0
 8           counter ← 0
 9           tasks ← searchAllTasks(appname, r)
10           if rank = TIMEEXECUTIONERROR
11             then for each task t in tasks
12                        do ▷ Each task 't' has an estimated time associated with it
13                           esttime ← estimatedTimeTask(t)
14                           acttime ← t.unsubmittedtime + t.pending + t.active
15                           acum = acum +abs(esttime − acttime)
16                           counter = counter +1
17                   resourcelist[r] ← acum/counter
18           if rank = EXPANSIONFACTOR
19             then for each task t in tasks
20                        do
21                           waittime ← t.active − t.pending
22                           exectime ← t.done − t.active
23                           acum = acum + (exectime+waittime)/exectime
24                           counter = counter +1
25                   resourcelist[r] ← acum/counter
26           failures ← 0
27           if rank = ESTIMATEFAILURERANK
28             then for each task t in tasks
29                        do esttime ← estimatedTime(t)
30                           if t.exitstatus = FAILED
31                             then failures ← failures +1
32                                  continue
33                             else acttime ← t.done − t.unsubmitted
34                                  if acttime ≤ esttime
35                                    then acum ← acum + (1 − esttime/acttime)
36                                    else acum ← acum + (1 − acttime/esttime)
37                                  counter ← counter +1
                    ▷ skewpercentage and failurepercentage represents a weight value.
                    ▷ skewpercentage + failurepercentage = 1
38                  resourcelist[r] ← acum/counter (1 − skewpercentage) + failures/size(tasks) (1 − failurepercentage)
39 return resourcelist
```

Figure 4.14. Pseudo-code of different ranking schemes implemented in Gridjobs.

grid initiative to mainly integrate computational resources from the Pacific area. PRAGMA is classified as a computational grid to exhibit a P2P-based management scheme. Different from other operational grids, PRAGMA does not deploy any reservation policy or mechanism to restrict resources utilization. It does mean that any PRAGMA user to require a computational resource needs to compete with other users for its utilization.

In this chapter we have discussed our approach to resource management in large scale grid systems such as PRAGMA. The problem is divided into two sub-problems: state estimation and decision making. For state estimation, Gridjobs employs statistical analysis to derive probabilistic models to estimate the resources behavior. Gridjobs divides the decision making problem in two sub-problems resource ranking and task division. Gridjobs implements ranking schemes to consider number of failures, enqueued time, execution time and resources load. The task division intends to divide the load amongst the participating resources in such way that all resources end their executions at the same time.

Gridjobs offers an extendable platform in which current technological trends could be implemented. Due to the Groovy versatility it is possible to derive a Domain Specific Language(DSL) inside to Gridjobs which provides primitives to leverage the development of programs to follow a Map-Reduce approach. On the other hand, Gridjobs only has tackled the resource performance estimation problem. However, the network bandwidth need to be considered in order to provide a complete computational framework.

CHAPTER 5

Experimental Results

For testing purposes, Gridjobs has been deployed on five computational assets belonging to the Pacific Rim Application and Grid Middleware Assembly (PRAGMA) infrastructure. Most of the PRAGMA assets have installed Globus Toolkit (GT) as their grid middleware by default. Hence, Gridjobs only interacts with GT based computational grids. However, due to its modular design, Gridjobs could be easily integrated with other grid middlewares such as GLite.

Although PRAGMA has approximately 30 members, many of them present different conditions to limit the utilization of the computational resources. During our experimental phase, we have tested the availability of the GRAM service executing a dummy globus command("*globus-job-run* **host***/jobmanager-sge /bin/hostname*") to all PRAGMA resources. Most of the resources returned errors which have been summarized in Table 5.1. GRAM 7 indicates authentication failures with the remote server. These failures could be caused because either DNS could not resolve correctly an IP address or the user account is not in the *grid-mapfile*. GRAM 12 is an error to indicate the gatekeeper is not running, the host is not reachable or the gatekeeper is on a non-standard port. GRAM 93 means that a LRM is not available on the server. More precisely, the grid middleware has not been integrated with any LRM instance. Finally, some resources did not present any problems at grid middleware but exposed connection problems via SSH protocol.

Error Code	Institution Name	Server
GRAM Error 7	JLU	grid1.jlu.edu.cn
GRAM Error 12	UChile	syntagma.dim.uchile.cl
	HKU	pragma1.grid.hku.hk
	OSAKU	cafe01.exp-net.osaka-u.ac.jp
		tea01.exp-net.osaka-u.ac.jp
	UTsukuba	bruce0.omni.hpcc.jp
	KISTI	jupiter.gridcenter.or.kr
	MIMOS	nucleus.mygridusbio.net.my
	APAC	myproxy.pragma-grid.net
	UNAM	malicia.super.unam.mx
	BeSTGRID	ng2hpc.canterbury.ac.nz
	ASGC	pragma001.grid.sinica.edu.tw
	BU	pop.cs.binghamton.edu
	HCMUT	supernode2.hcmut.edu.vn
	HUT	bkluster.hut.edu.vn
	IOIT-HCM	venus.ioit-hcm.ac.vn
	MU	mahar.csse.monash.edu.au
	CNIC	pragma.sdg.ac.cn
GRAM Error 93	CUHK	server1.itsc.cuhk.edu.hk
	NECTEC	grid64.hpcc.nectec.or.th
	NCHC	nacona00.nchc.org.tw
	LZU	pragma.lzu.edu.cn
SSH Error	USM-CS	aurora.usmgrid.myren.net.my
	USM-Ph	hawk.usm.my
	IHPC	sirius.ihpc.a-star.edu.sg
	UZH	ocikbpra.uzh.ch

Table 5.1. Errors observed from *jas@komolongma.ece.uprm.edu*.

5.1 UPRM in PRAGMA

The University of Puerto Rico at Mayagüez (UPRM) shares two clusters *volatile.ece.uprm.edu* and *komolongma.ece.uprm.edu*. *komolongma* is a cluster x86-based. It has deployed the penultimate release of the Rocks system. In addition, VDT packages have been also integrated with the system. VDT installs all the components required by a computational grid asset such as security modules, gatekeepers and scripts to integrate the grid middleware with a LRM instance. In fact, prepare a cluster for sharing it with PRAGMA would not take more that two days. On the other hand, *volatile* is a cluster Itanium-based. This system presents important updating problems of different system programs. For instance, Rocks is the *volatile* operating system but current Rocks releases are not available for Itanium systems. Similarly, other programs such as compilers and grid middlewares, are not available for recent versions. Despite many components provide their source code, their compilation many times require to hack the source code.

5.2 Testbed

Gridjobs was successfully deployed over five PRAGMA clusters: *fsvc001.asc.hpcc.jp*, *sakura.hpcc.jp*, *rocks-52.sdsc.edu*, *rocks-153.sdsc.edu* and *komolongma.ece.uprm.edu*. All those systems have installed Globus Toolkit and Sun Grid Engin (SGE) as LRM.

fsvc001 and *sakura* are clusters shared by the national institute of Advanced Industrial Science and Technology (AIST) of Japan. These systems contribute with 81 compute nodes, over 300 Gbytes of RAM and 6.3 Tbytes of storage. *rocks-52* and *rocks-153* are clusters shared by San Diego Super Computing Center. They contribute with 20 compute nodes, over 30 Gbytes of RAM and approximately 900 Gbytes of storage. Finally, *komolongma* contributes with 60 compute nodes, 60 Gbytes of RAM and 1.5 Tbytes of storage.

Late 2008, PRAGMA started to migrate toward a novel security structure. This new scheme supports the definition of Virtual Organizations. Many computational assets have started the migration to the new scheme but others keep the previous scheme because there are some technical details to require be addressed like to enhance the privacy levels amongst

the members under a same VO. *sakura* and *rocks-53* have adopted the new scheme. This adoption presents new challenges for the initial desing in Gridjobs. For instance, the new scheme does not support the legacy SSH protocol but the GSI-SSH protocol. GSI-SSH exhibits interesting enhancements over SSH such as single sing-on, file transfer service and credential forwarding. However, a detailed study to determine the impact of the new protocol over Gridjobs services like DeploymentService has not been carried out. Similarly, since the concept of user has disappeared, each computational asset perceives VO submitting jobs and does not discriminate by user. This fact requires to analyse how the Gridjobs presumptions are now affected when the application perfomance analysis is carried out. Figure 5.1 shows the current testbed.

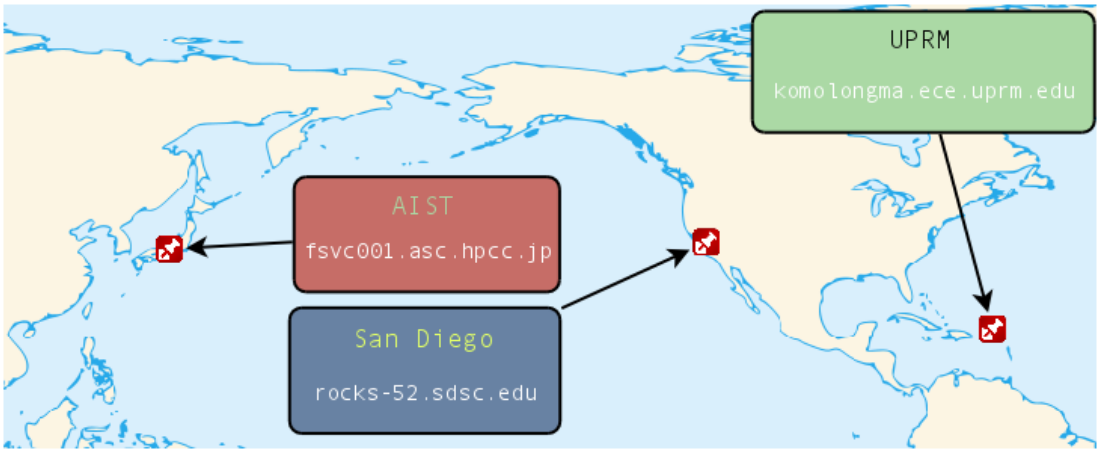

Figure 5.1. Assets employed during the experimental phase.

5.3 Application Deployment

Gridjobs has a deployment service able to install applications over computational grids. Gridjobs assumes that each computational resource exhibits an infrastructure similar to the infrastructure depicted in Figure 5.2. There are several Linux-based distributions to deploy this kind of infrastructures such as Rocks, ClusterKnoppix, ParallelKnoppix, CLIC, and Red Hat Cluster, among others. This configuration requires that the main storage device can be accessed from any cluster's compute node, even the headnode. Rocks and other cluster rocks-based distributions employ Network File System(NFS) protocol to supply a homogeneous file system

view. Thus, a user could access to her files indepedent on the compute node where she is logged in.

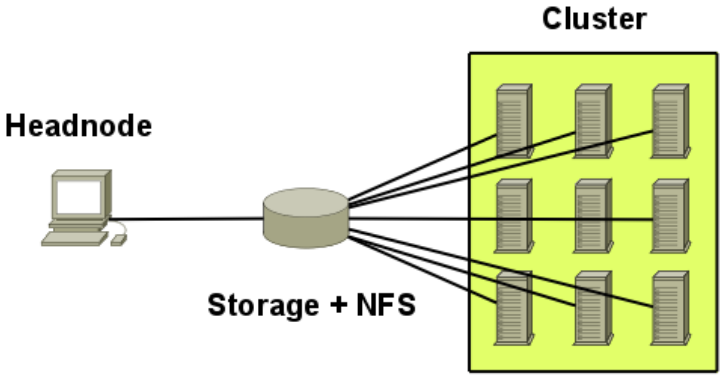

Figure 5.2. Cluster .

When the *ApplicationdeploymentService* is invoked for deploying an application, it contacts the cluster headnode and installs the application on it. Since, the availability of NFS is assumed, when the deployment process finishes, the user could now execute the application from any compute node in the cluster comanded by the aforementioned headnode. This characteristic is necessary because Gridjobs dispatches task execution requests to the headnode which passes the requests to the LRM. The LRM according to its own management policies and scheduling algorithms, decides which compute node would attend the execution request. If the application deployment is not visible from any cluster's compute node, the execution request would fail.

5.4 Setting Experiments

Gridjobs is a framework to operate over real computational grids. Different from simulated environments where each experiment takes minutes or few hours to run, experiments in Gridjobs could last from 24 up to 48 hours to run. Therefore, Gridjobs exhibits a minimal infrastructure to support the execution of long lasting experiments.

5.4.1 Tables for supporting experiments in Gridjobs

Gridjobs has defined two tables used to invoke multiple and sequential application executions.

Figure 5.3 shows the most important attributes of the *experiment* and *experimentinstance*

```
             Table "public.experiment"                        Table "public.experimentinstance"
   Column     |            Type                    Column     |            Type
--------------+-----------------------------   ---------------+-----------------------------
id            | bigint                          appname       | character varying(255)
version       | bigint                          endtime       | timestamp without time zone
endtime       | timestamp without time zone     highparam     | integer
max           | bigint                          lowparam      | integer
nexttoexecute | bigint                          sequence      | bigint
sequence      | bigint                          starttime     | timestamp without time zone
starttime     | timestamp without time zone     stride        | integer
```

Figure 5.3. Tables to support the tracking process of experiments in Gridjobs.

tables. The *experiment* table stores the information associated with the number of executions required in a given experiment. The *max* attribute indicates the maximum number of executions that an experiment expects to execute. *nexttoexecute* is a pointer to the next experiment to be executed. This number is directly associated with the *sequence* attribute of the *experimentinstance* table.

The attributes of the *experimentinstance* table are presented in the right side of Figure 5.3. This table contains all details required for executing an application under Gridjobs. Figure 5.4 presents possible entries for the aforementioned tables. The *experiment* entry says that the next experiment to be executed is the number 88 and the maximum number of executions in this experiment is 107. On the other hand, the *experimeninstance* entry presents the values associated with execution 88. In this case, the application to be executed is *ls4_v02* and the range of values that this application would process is from 0 up to 200.

```
experiment entry
 max | nexttoexecute | sequence
-----+---------------+----------
 107 |            88 |        1

experimentinstance entry
 appname | endtime | highparam | lowparam | sequence | starttime | stride
---------+---------+-----------+----------+----------+-----------+--------
 ls4_v02 |         |       200 |        0 |       88 |           |      1
```

Figure 5.4. Sample entries of *experiment* and *experimentinstance* tables.

5.4.2 Experiment Workflow

Figure 5.5 depicts a flow diagram to indicate how the experiments are carried out. Initially, Gridjobs requires to make dummy executions of the application under evaluation. These executions are carried out on the available computational resources. The *Statistical* service is

then invoked. When this service finishes its execution, three different R scripts are available for estimating the resource performance. (One script models the UNSUBMITTED state, other script the PENDING state and a third script models the ACTIVE state). Now, the *Experiment* service can be invoked. The *experiment* table is queried, if the *nexttoexecute* value is greater than *max*, the experiment finalizes. Otherwise, the *experimentinstance* table is queried and the corresponding application invocation is retrieved. Gridjobs then executes its ranking and scheduling algorithms and distributes the application load amongst the available resources. Gridjobs estimates a finalization time and schedules an application monitor. On the estimated finalization time, the monitor is fired and it determines if all the submitted tasks have finalized. When they finalize, Gridjobs invokes the 'Statistical' service to update the R scripts in order to reflect the possible changes observed in the last execution. The aforementioned process is repeated while *nexttoexecute* is less or equal than the *max* value.

5.5 Results

5.5.1 Resource behavior is barely represented by a unique probalistic function

Different works have modeled the resources behavior through different approaches such as data mining techniques, heuristics and probability functions. A classical approach is to do an offline statistical analysis over historical data, then to infer resources behavior over long periods of time. The models deducted are then integrated to simulated environments where different conditions are controlled but these environments hardly could represent real infrastructures. These models barely reflect the possible presence of competing applications to content for resources. In front of rival computational intensive applications, models lost validity. Data collected and graphics derived during our experiments show the presence of important levels of variability over three PRAGMA resources.

rocks-52.sdsc.edu

Figure 5.6 presents the *rocks-52* behavior perceived by Gridjobs at *komolongma*. The leftmost sub-graph shows the elapsed time on UNSUBMITTED state. This sub-graph shows that most of the data are concentrated around of five seconds, but also an important amount of

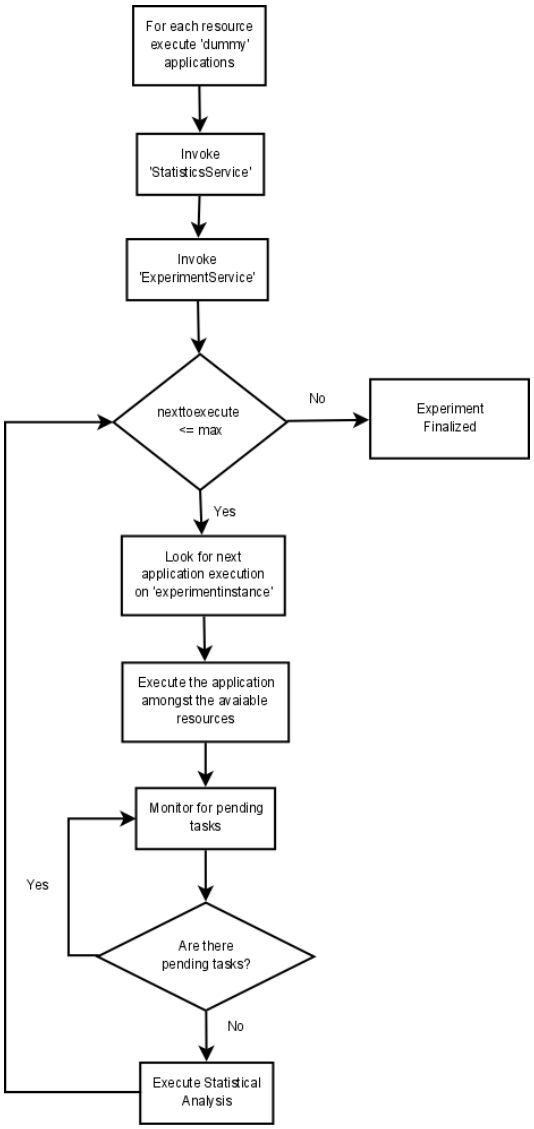

Figure 5.5. Diagram Flow to describe how the experiments are executed.

observations are found between five and ten seconds. The middle sub-grapgh indicates that most of the requests lasted twenty five seconds in PENDING state. However, there were also observed other requests lasting up to four minutes. In fact, longer periods of time could be possible but Gridjobs allows to the user to define hard limits to restrict the maximum time that a task could last. Finally, the rightmost sub-graph shows that most of the executions lasted five or six minutes. However, unexpected values around of two minutes were also registered. This fact is algorithmically impossible but unfortunately in rare ocassions a GRAM instance could wrongly indicate that task has "successfully" finished. A non-invasive framework like Gridjobs, could hardly determine if an information provided by a GRAM service is right or wrong. Hence, additional modules to corroborate the information provided by GRAM services are necessary. At this time, Gridjobs relies on the information provided by the remote grid information services. Table 5.2 shows different probability functions selected by Gridjobs during

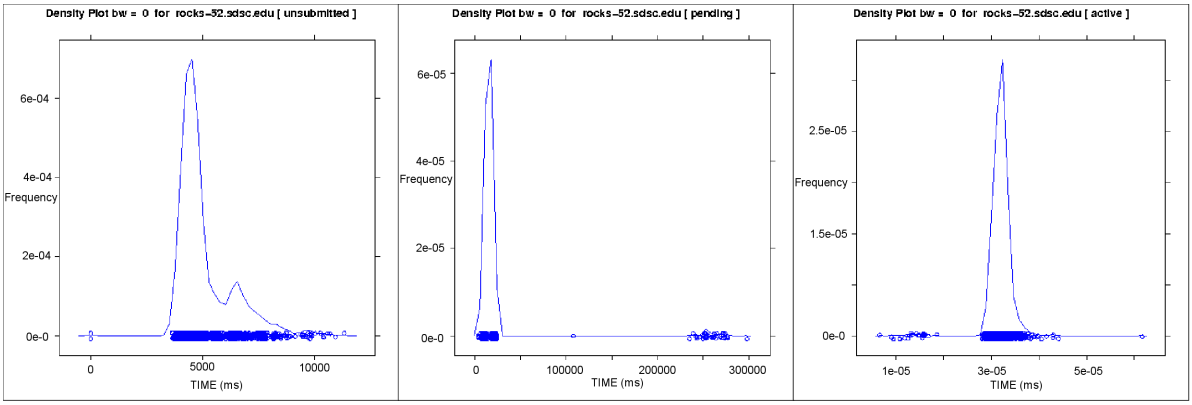

Figure 5.6. rocks-52.sdsc.edu observed behavior.

a serie of executions carried out on *rocks-52*. In this particular case it is possible to see how the probability function varies to model the ACTIVE stage.

fsvc001.asc.hpcc.jp

fsvc001 is a resource located on Japan. Similarly to other PRAGMA resources, the inter-connectivy is through Internet. Figure 5.7 presents the behavior observed in *fsvc001*. The

Date Time	Probability Function	Function Parameters
2009-05-19T16:06:39.007-04:00	rlnorm	meanlog=12.1789817289487 sdlog=0.14
2009-05-19T16:30:34.380-04:00	rCauchy	location=180161.49 scale=13461.54
2009-05-19T16:57:07.789-04:00	rlogis	location=180100.78 scale=11333.83
2009-05-19T17:21:21.452-04:00	rlogis	location=179736.02 scale=10261.76
	...	
	...	
2009-05-19T21:28:20.343-04:00	rexp	rate=5.64453549513913e-06
2009-05-19T21:55:49.918-04:00	rexp	rate=5.64678430112628e-06
2009-05-19T22:23:23.580-04:00	rexp	rate=5.63776552035679e-06
2009-05-19T22:58:48.770-04:00	rCauchy	location=180081.35 scale=249.50
	...	
	...	
2009-05-20T02:31:26.425-04:00	rexp	rate=5.71338576067818e-06
2009-05-20T03:12:07.691-04:00	rCauchy	location=178382.99 scale=5489.69
	...	
	...	
2009-05-20T09:07:31.565-04:00	rCauchy	location=178788.37 scale=5328.06
2009-05-20T09:31:43.100-04:00	rCauchy	location=178800.02 scale=5234.30
2009-05-20T09:57:04.171-04:00	rCauchy	location=178718.77 scale=5341.79
	...	
	...	

Table 5.2. Some probability functions to model the ACTIVE stage on *rocks-52*.

leftmost sub-graph presents an important variability on UNSUBMITTED stage. In fact, most of the requests last less than five seconds, but times around of the ten seconds have also occurred. That is because Internet does not provide any QoS support for packets interchange. The PENDING state, middle sub-graph, exhibits a stable behavior. It does mean, during this period of time in particular, the fsvc001's LRM had few enqueued tasks. This circumnstance allows that arriving tasks were promptly attended. Similarly, the right-most subgraph shows that many of the executions were around of the six and seven minutes but unexpected low times were also observed. Finally, note that *fsvc001* is one of the largest clusters shared on PRAGMA. Despite the SCMSWeb monitoring tool informs that this computational resource counts with more than 250 computational nodes, the management policies establish a reduced number of nodes available to the PRAGMA users.

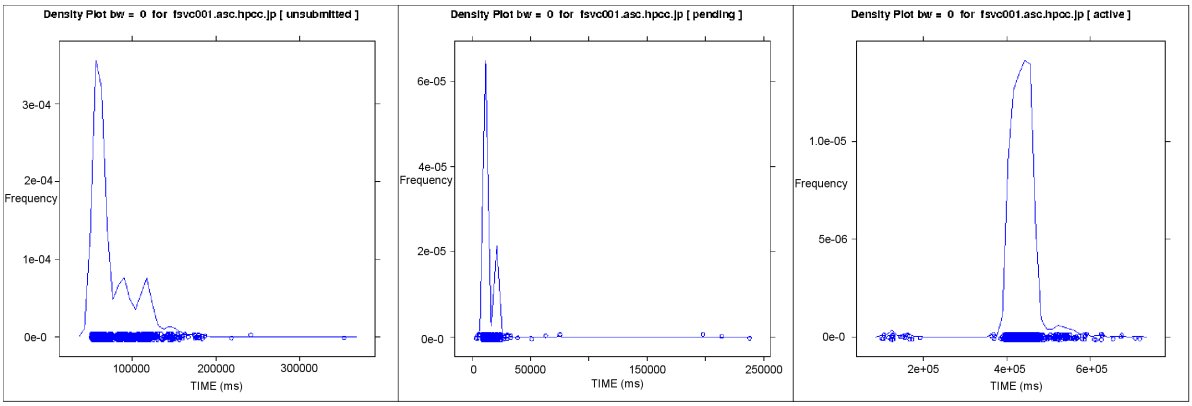

Figure 5.7. fsvc001 observed behavior.

komolongma.ece.uprm.edu

komolongma observes the most predictable behavior, Figure 5.8. For instance, many tasks lasted in UNSUBMITTED stage from zero to four seconds. Similarly, on PENDING stage, several tasks lasted less than five seconds. Finally, on execution time the time ranged between one minute and two. This remarkable difference on uncertainty performance could be justified by different reasons. First, Gridjobs has been deployed on *komolongma*. Thus, network latency

Date Time	Probability Function	Function Parameters
2009-05-19T16:06:39.007-04:00	rCauchy	location=114287.00 scale=22.81
2009-05-19T16:30:34.380-04:00	rCauchy	location=114010.34 scale=516.74
2009-05-19T16:57:07.789-04:00	rCauchy	location=114318.68 scale=100.99
2009-05-19T17:21:21.452-04:00	rCauchy	location=114116.08 scale=455.15
	...	
	...	
2009-05-19T21:28:20.343-04:00	rCauchy	location=113248.47 scale=55.53
2009-05-19T21:55:49.918-04:00	rexp	rate=8.7970393132501e-06
2009-05-19T22:23:23.580-04:00	rCauchy	location=113249.01 scale=73.81
2009-05-19T22:58:48.770-04:00	rexp	rate=8.77195981549397e-06
	...	
	...	
2009-05-20T02:31:26.425-04:00	rCauchy	location=113252.31 scale=92.91
2009-05-20T03:12:07.691-04:00	rCauchy	location=113252.62 scale=95.56
	...	
	...	
2009-05-20T09:07:31.565-04:00	rCauchy	location=113269.93 scale=269.87
2009-05-20T09:31:43.100-04:00	rCauchy	location=113263.77 scale=225.54
2009-05-20T09:57:04.171-04:00	rCauchy	location=113273.34 scale=272.52
	...	
	...	

Table 5.3. Some probability functions to model the ACTIVE stage on *fsvc001*.

is not an issue because the communication between Gridjobs and komolongma does not require network intervention. Second, different from the previous resources, *komolongma* presents a low computational demand and absence of management policies. Low computational demand then implies that LRM's queue is mostly empty in such a way that as soon as one request arrives, it is dispatched to some compute node. Finally, the lack of management policies provides a computational environment where user computational demands are fully met. These facts reduce the uncertainty levels that Gridjobs could observed.

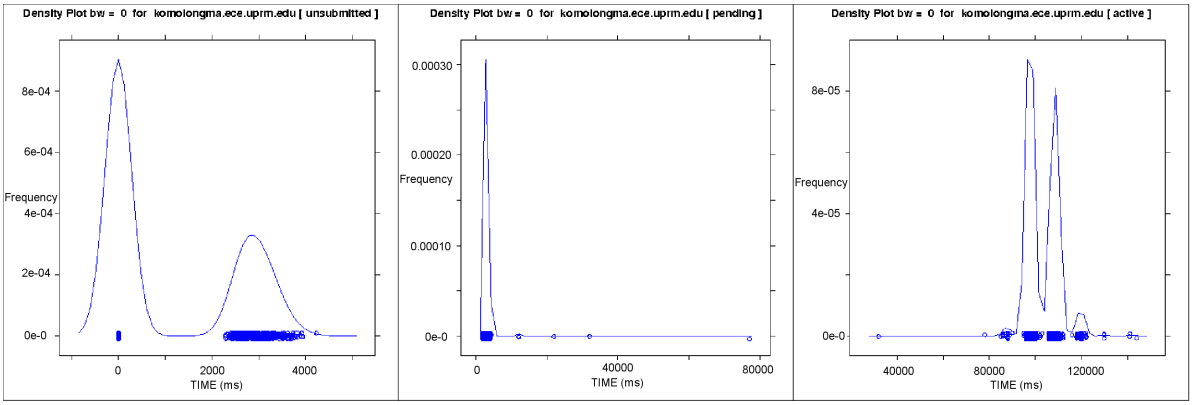

Figure 5.8. komolongma observed behavior.

5.5.2 Run-time Probability Function Selection

The following set of experiments were designed to evaluate the probabily function selection process on runtime. The characteristics of the experiments are described in Table 5.5. In these experiments we considered two ranking schemes: *defaultrank* and *gwrank*. These rankings establish an order in which the resources are sorted. For instance, *defaultrank* sorts the resources according to the resource performance. Thus, the fastest resource goes first and the slowest goes last. On the other hand, *gwrank* does not consider the resource performance but the resource load. Thus, the resource exhibiting the lightest load goes first and the resource with the heaviest load goes last.

Date Time	Probability Function	Function Parameters
2009-05-19T16:06:39.007-04:00	rlogis	location=240788.79 scale=7859.96
2009-05-19T16:30:34.380-04:00	rnorm	mean=242170.21 sd=14579.02
2009-05-19T16:57:07.789-04:00	rCauchy	location=239998.96 scale=10.72
2009-05-19T17:21:21.452-04:00	rCauchy	location=239999.05 scale=9.78
	...	
	...	
2009-05-19T21:28:20.343-04:00	rCauchy	location=239998.20 scale=6.64
2009-05-19T21:55:49.918-04:00	rCauchy	location=239997.70 scale=9.63
2009-05-19T22:23:23.580-04:00	rCauchy	location=239997.89 scale=9.46
2009-05-19T22:58:48.770-04:00	rCauchy	location=239997.80 scale=9.16
	...	
	...	
2009-05-20T02:31:26.425-04:00	rexp	rate=4.24440652317988e-06
2009-05-20T03:12:07.691-04:00	rexp	rate=4.23943505263254e-06
	...	
	...	
2009-05-20T09:07:31.565-04:00	rCauchy	location=238057.47 scale=4547.83
2009-05-20T09:31:43.100-04:00	rexp	rate=4.30649097027218e-06
2009-05-20T09:57:04.171-04:00	rexp	rate=4.30525063302825e-06
	...	
	...	

Table 5.4. Some probability functions to model the ACTIVE stage on *komolongma*.

Id	Initial Time	Final Time	Ranking	Num. Executions
1	May-19T15:42	May-21T15:18	Default	100
2	May-26T18:19	May-27T15:17	Default(F)	50
3	May-25T11:50	May-26T10:17	GridWay	50
4	May-25T09:07	May-29T06:41	GridWay(F)	50

Table 5.5. Experiments executed between May 19 to May 29.

rocks-52.sdsc.edu

For experiment 1 the more accurate p.f. was *Cauchy*. In our analysis we have considered over estimations and under estimations. The over estimation occurs when the actual execution time is less than the estimated time. On average, it was found an over estimation of 12%. On the other hand, the under estimation was approximately 10%. Gridjobs employed this function on 76% of the times to predict the resource behavior. The parameters of this function are depicted in Figure 5.9. The left subgraph suggests that the location parameter exhibited a value close to 180000. On the other hand, the subgraph on the right side suggests a scale parameter concentrated around 5000.

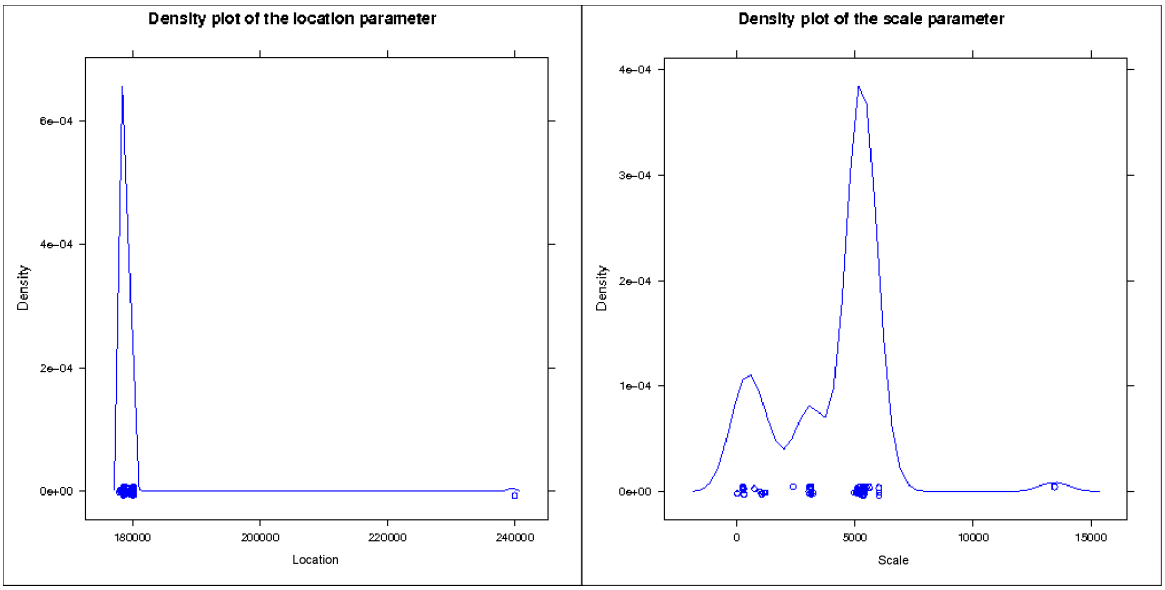

Figure 5.9. Density plot of p.f. to model the ACTIVE stage on *rocks-52* for experiment 1.

For experiment 2, the more accurate p.f. was again *Cauchy*. In this experiment, Gridjobs employed Cauchy over 68% of the times. When this function made its estimations, its predictions, on average, over estimated 7.8% of the actual time. The under estimated value was 13%. The function parameters are depicted in Figure 5.10. The density functions in the aforementioned figure, suggests a location value close to 180000 and the scale value most of the time was around 5000. Lognormal p.f. was other function employed by Gridjobs to model the resources over 20% of the times.

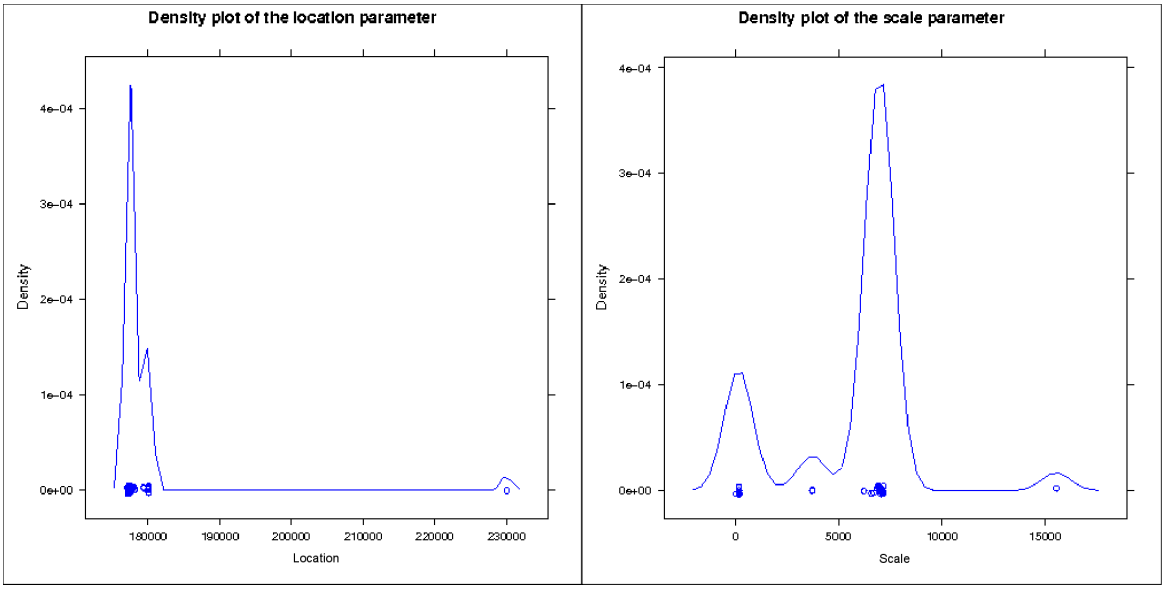

Figure 5.10. Density plot of p.f. to model the ACTIVE stage on *rocks-52* for experiment 2.

For experiment 3, the more accurate p.f. was the *Logistic* distribution. In this experiment, Gridjobs employed Logistic over 47% of the times. The function parameters are depicted in Figure 5.11. This time, the density function for the *location* parameter of the Logistic distribution suggests a value betweer 164000 and 167000. The scale parameter(right-hand subgraph) exhibited a value between 6000 and 6500. It is worthwhile to remark the following facts. First, this experiment, during seven hours approximately, does not consider the *komolongma* resource. Fictitious problems were created in such a way that GRAM service on *komolongma* was unable to process any task directed to *komolongma*. Second, 47% seems a poor prediction, however, other distributions such Cauchy(location=168000,scale $\approx$ 6000) and Lognormal(mean=12.02,sd[1]$\approx$ 0.07) were also selected 21% and 25% of the times, respectively. Thus, Gridjobs predicted the resource behaviour with an important accuracy degree over 92% of the times. For experiment 4, the more accurate p.f. was the *Logistic* distribution. It was used to predict on 45% of the times. Figure 5.12 suggest the location (approx. 164000 - 165000) and scale (between 6000 and 7000) parameter values. In this particular experiment, the *Cauchy* function exhibited a poor prediction when it is compared with previous experiments (approx. 2 minutes of difference between estimated and actual times). However, other p.f.

[1]standard deviation

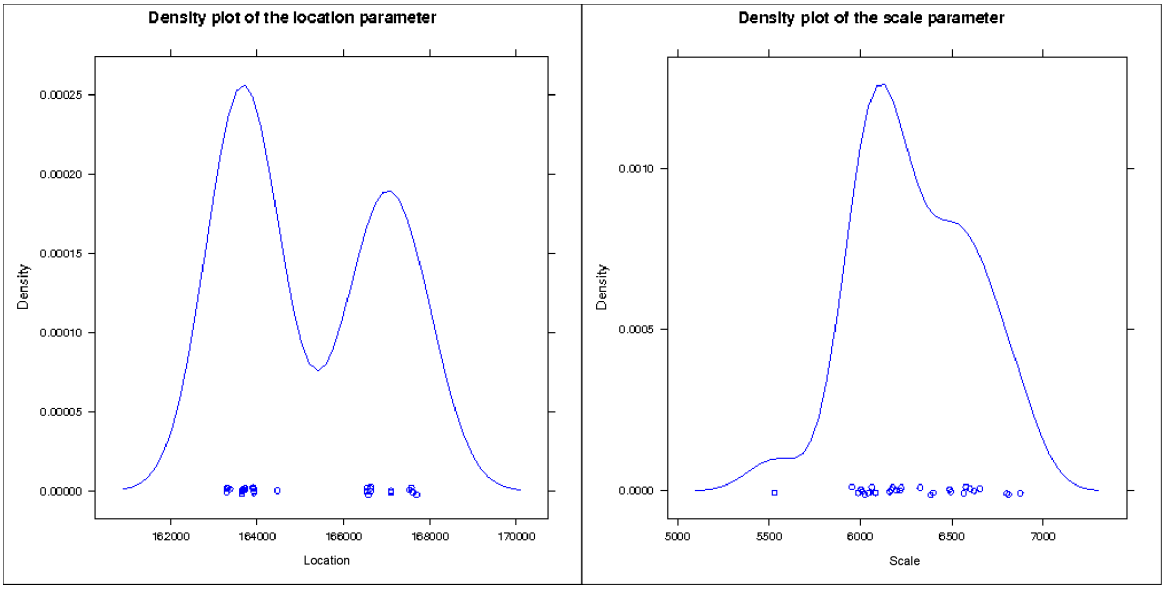

Figure 5.11. Density plot of p.f. to model the ACTIVE stage on *rocks-52* for experiment 3.

such as normal and log normal were also selected to predict resources performance. Thus, Gridjobs made an acceptable prediction over 69% of the times, under a faulty scenario.

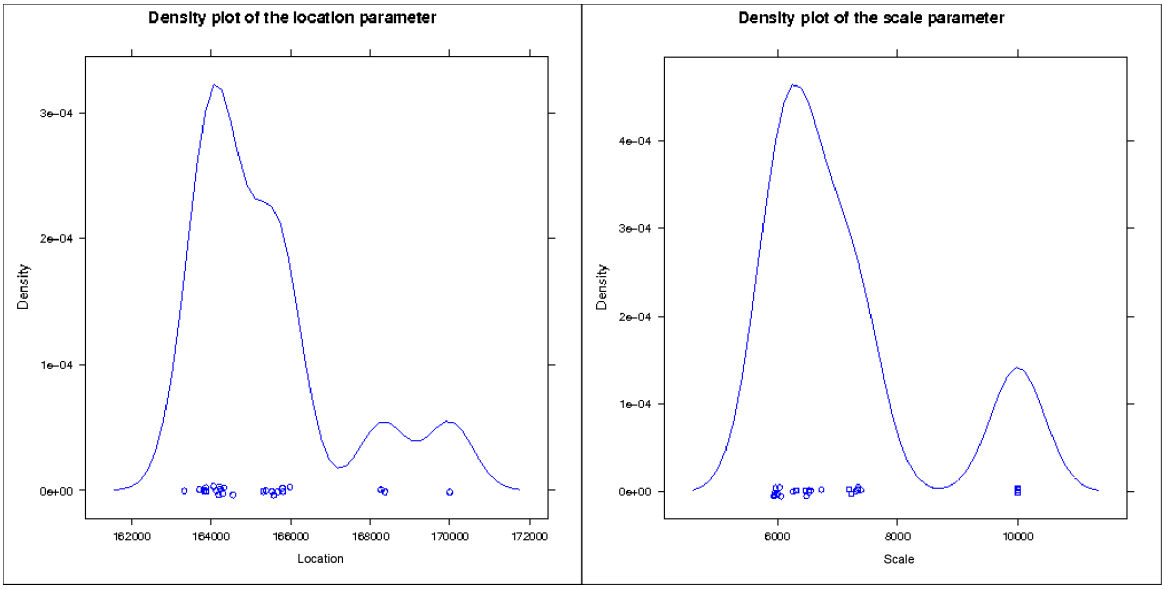

Figure 5.12. Density plot of p.f. to model the ACTIVE stage on *rocks-52* for experiment 4.

fsvc001.asc.hpcc.jp

On Experiment 1, Gridjobs selected four p.f. (Weibull, Cauchy, Logistic, Exponental) to predict the compute nodes performance on *fsvc001*. Cauchy, Weibull and Logistic exhibited

important levels of accuracy. In particular, *Cauchy* p.f. was selected 84% of the times and was also the most accurate p.f. Figure 5.13 presents two density plots to suggest the values of location and scale parameters. The location paramter presented a value close to 113200 and the scale parameter had a value close to 150. In general, Gridjobs over 93% of the times selected a p.f. to do a good work of prediction.

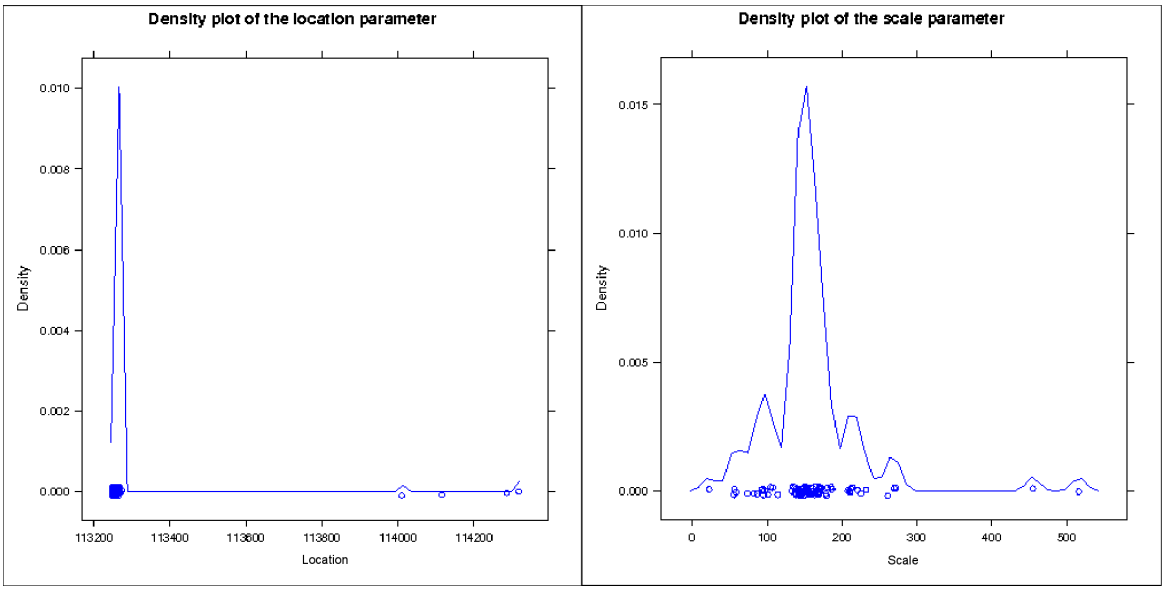

Figure 5.13. Density plot of p.f. to model the ACTIVE stage on *fsvc001* for experiment 1.

On Experiment 2, Cauchy was again the more selected p.f. with 49% of the times. Similarly, the Logistic p.f. showed a good level of accuracy and was choosed 16% of the times. However, the Exponential function was used to model the performance of this resource over 30% of the times but it had a poor level of accuracy. In general, it is possible to say that over 65% of the times, Gridjobs made a good prediction. Figure 5.14 depicts density functions of location and scale parameters of the Cauchy p.f. used in the experiments. For the location parameter a value close to 113200 and an approximated value to 200 was observed for the scale parameter.

On Experiment 3, Gridjobs tried with several p.f. with important levels of accuracy such as Gamma, Weibull, Log Normal, Poisson, Cauchy and Logistic. Despite Cauchy was selected over 72% of the times, this was not the more accurate p.f. Logistic made the best prediction

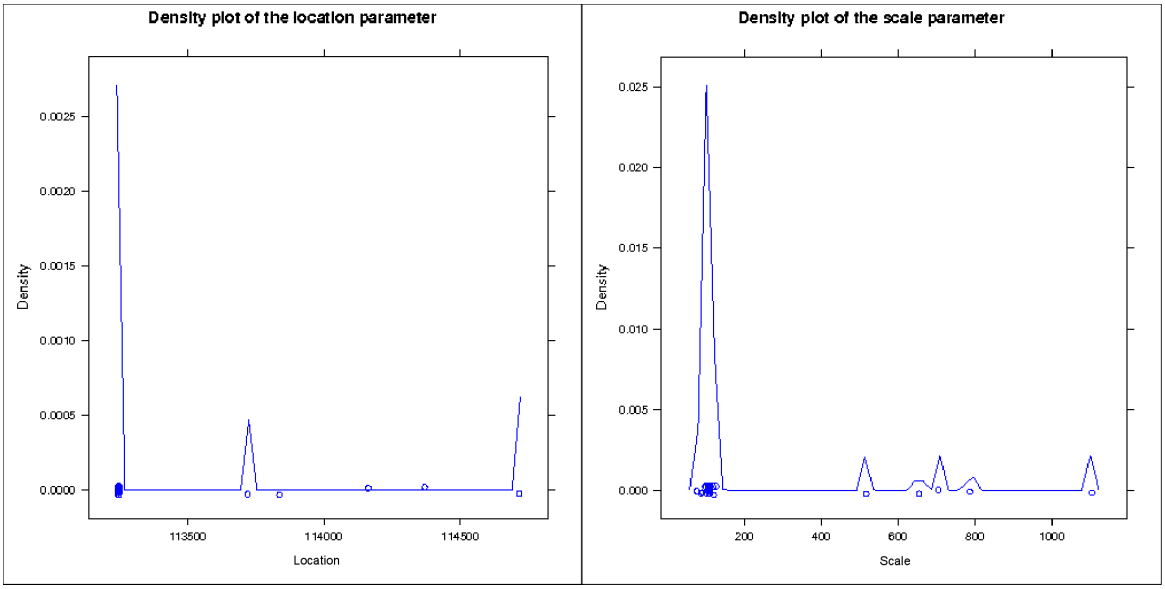

Figure 5.14. Density plot of p.f. to model the ACTIVE stage on *fsvc001* for experiment 2.

and was employed to predict the *fsvc001* resources performance 15% of the times. The parameters of this p.f. are graphically represented on Figure 5.15. The Cauchy parameter values were approximately 115200 and over 180 for location and scale parameters, respectively. Finally, for Experiment 4, only two p.f. were selected by Gridjobs. Again, Cauchy exhibited a

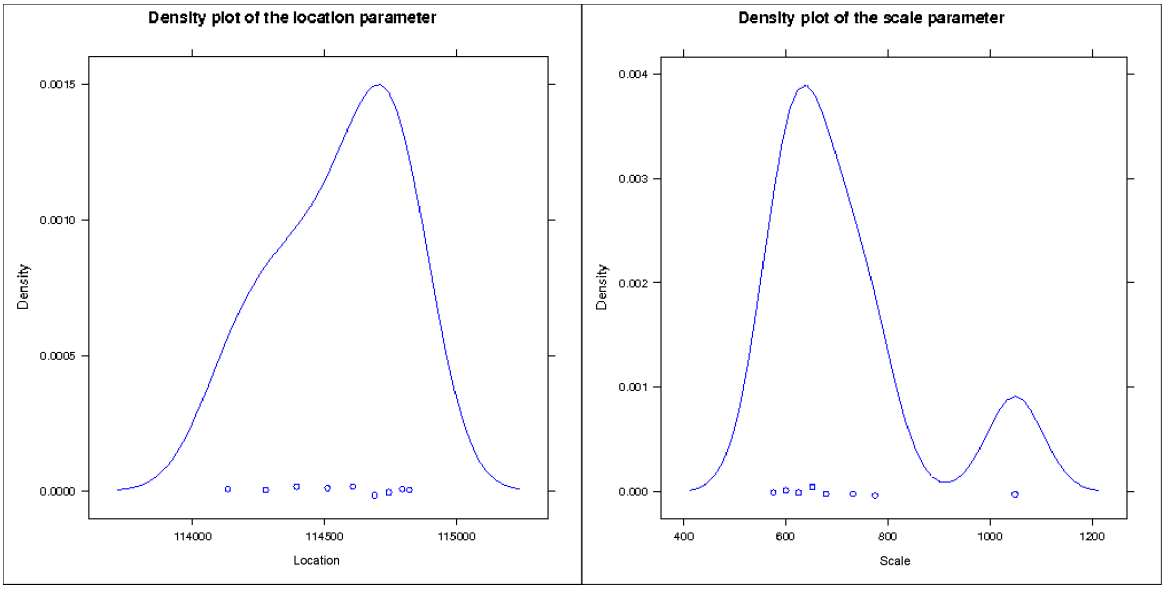

Figure 5.15. Density plot of p.f. to model the ACTIVE stage on *fsvc001* for experiment 3.

great accuracy and was used to model the resources performance 98% of the times. Figure

5.16 presents the density functions corresponding to the location and scale parameters.

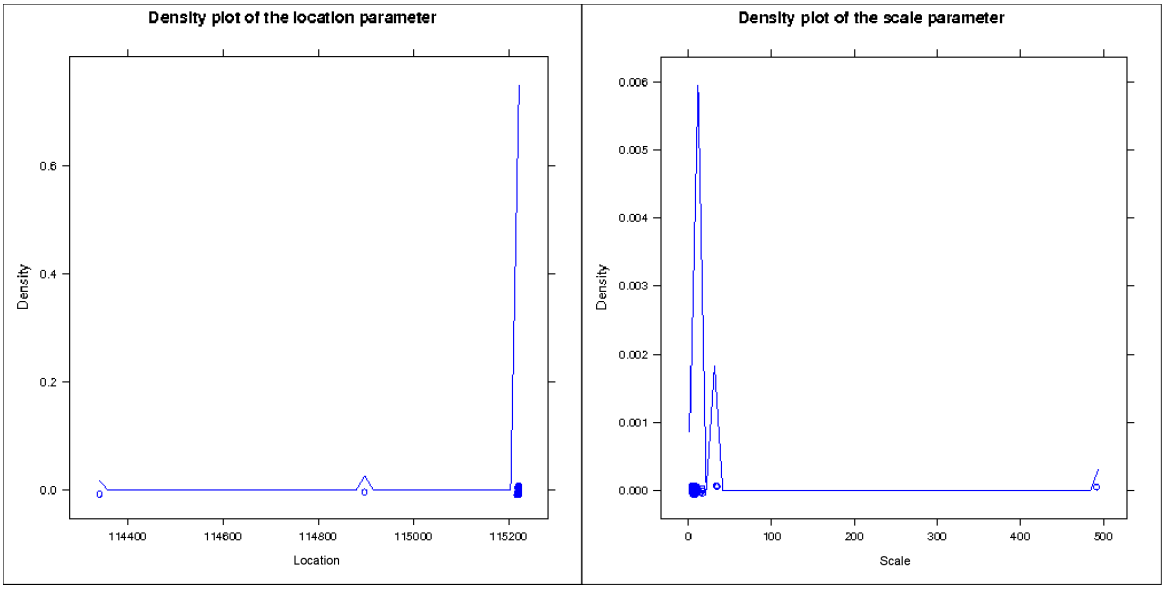

Figure 5.16. Density plot of p.f. to model the ACTIVE stage on *fsvc001* for experiment 4.

komolongma.ece.uprm.edu

On Experiment 1, Grgidjobs exhibited a misleading behavior. Two p.f. were the most selected functions to predict the komolongma resources performance, Exponential and Cauchy. The Exponential p.f. got 44%, while Cauchy was selected on 52% of the times. However, Exponential, with a rate parameter equal to 4.30 approximately, showed a very poor accuracy level. On the other hand, Cauchy got the best accuracy level amongst other selected p.f. such as Logistic, Normal and Exponential. Figure 5.17 suggests 290000 and 4000 to the location and scale parameters, respectively.

During Experiment 2, where *komolongma* was off-line for eight hours approximately, Cauchy was selected by Gridjobs to model the *komolongma* performance over 94% of the times. In addition, Cauchy made the best prediction. Figure 5.18 suggests the approximated values to location and scale parameters are 240000 and 0, respectively.

During Experiment 3, Gridjobs selected three p.f. Cauchy, Logistic and Normal. When Cauchy and Logistic functions were selected, the estimated time overpassed the actual time by over one minute. Gridjobs selected Cauchy 94% of the times when a modeling function was

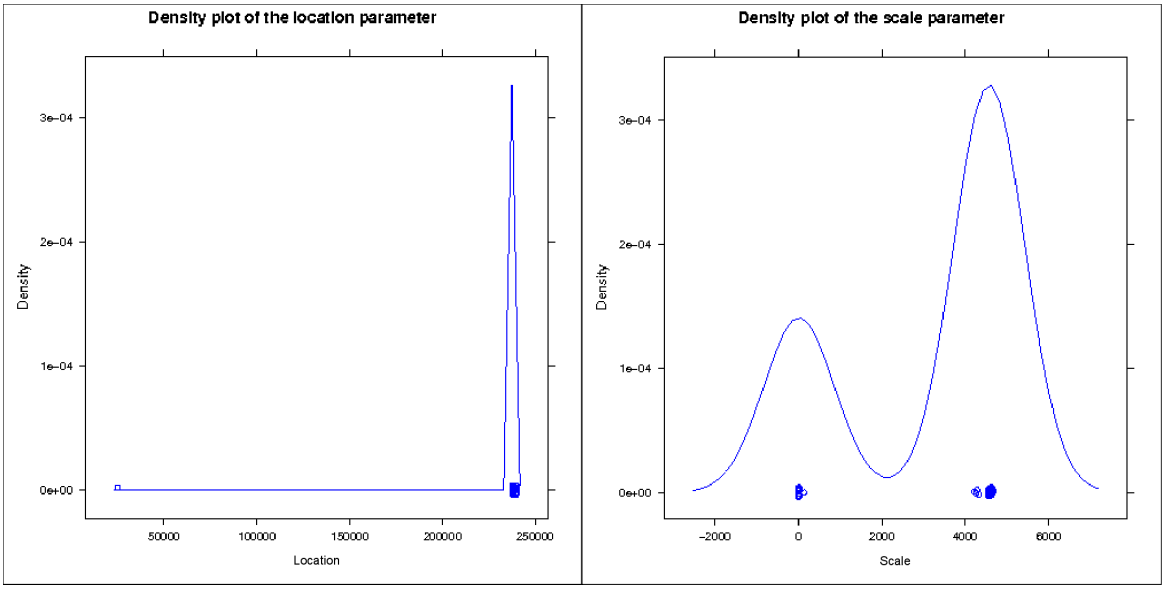

Figure 5.17. Density plot of p.f. to model the ACTIVE stage on *komolongma* for experiment 1.

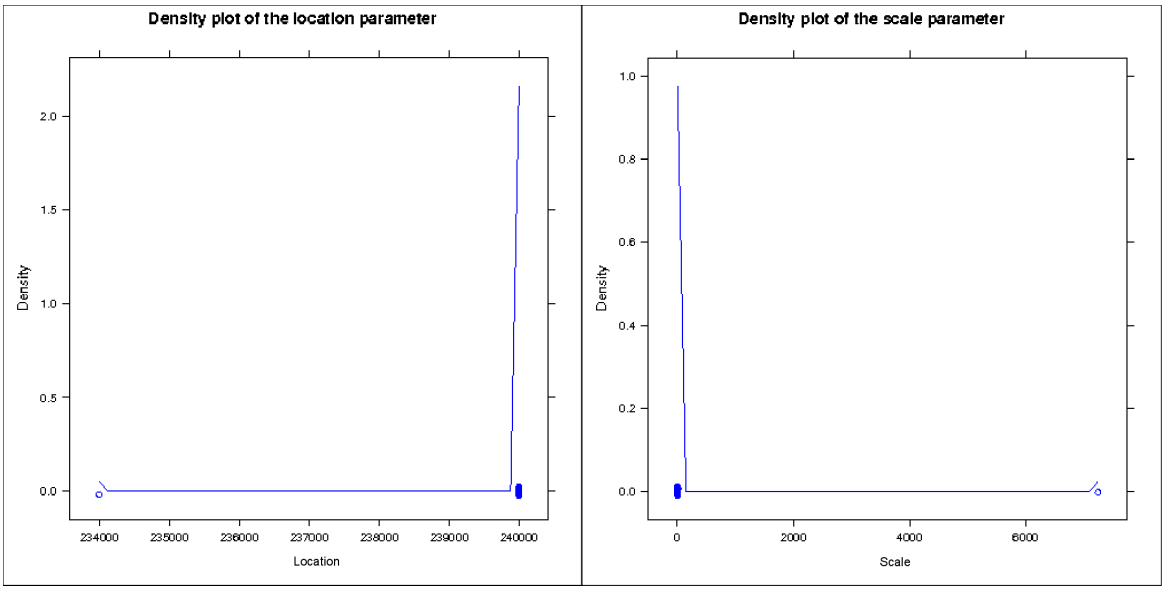

Figure 5.18. Density plot of p.f. to model the ACTIVE stage on *komolongma* for experiment 2.

required. However, the more accurate function was the Normal p.f. (mean equal to 230143 and standard deviation over 38900). Figure 5.19 presents the density plots for the location and scale parameters of the Cauchy p.f. The location parameter values are between 239980 and 239990.

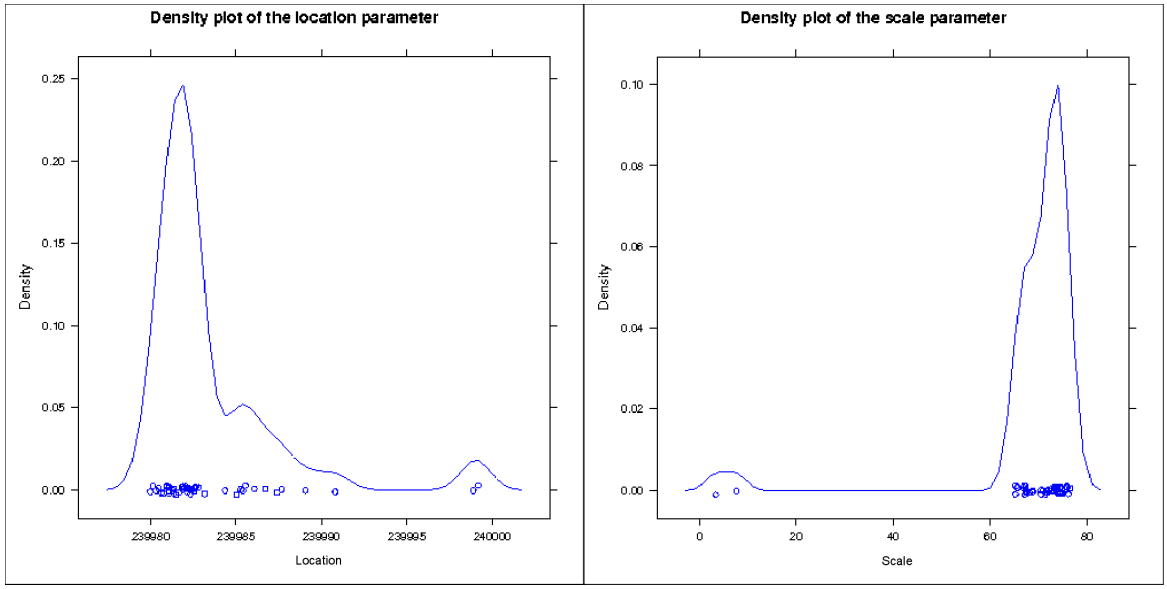

Figure 5.19. Density plot of p.f. to model the ACTIVE stage on *komolongma* for experiment 3.

Finally, Experiment 4 showed that Gridjobs selected amongst three p.f. Gamma, Weibull and Cauchy. Cauchy was employed to model the resource over 95% of the times. Figure 5.20 suggests the values of location and scale parameters.

5.5.3 Minimum Amount of Data Required to Forecast

Gridjobs relies on previous performance observations in order to predict the performance of coming requests. Under experimental conditions, this analysis is carried out over all observed data. However to carry out this kind of analysis over large sets of data could be a demanding computational task. In this experiment, three scenarios were under testing in order to determine if there is an "ideal" number of executions that could correctly model a resource performance. For "ideal" we meant the minimal number of previous executions to allow to model all resources with acceptable levels of accuracy.

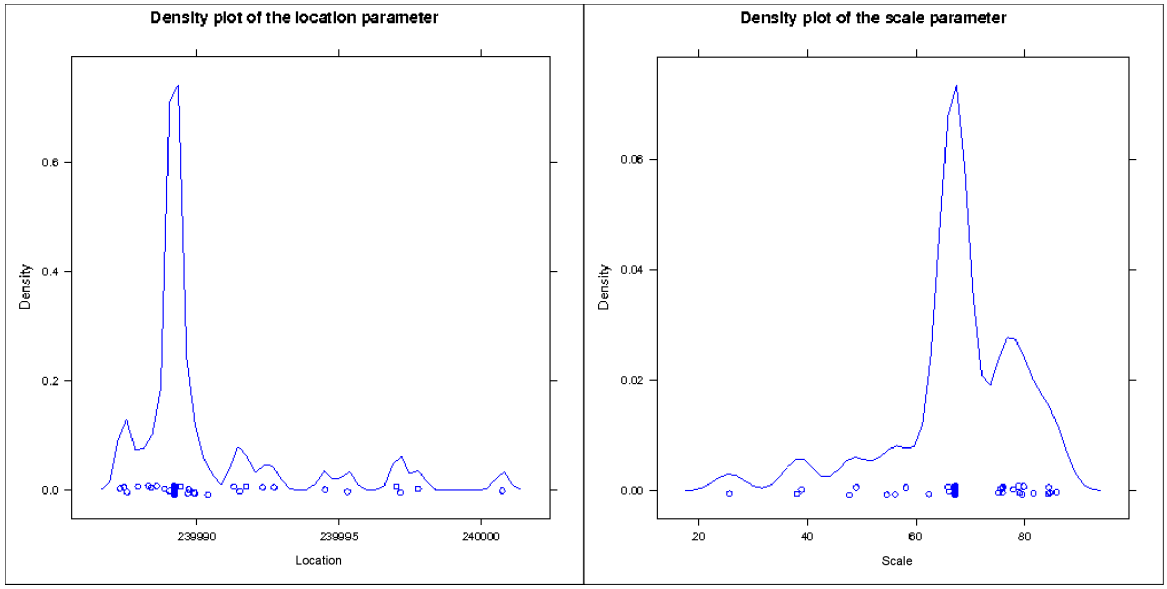

Figure 5.20. Density plot of p.f. to model the ACTIVE stage on *komolongma* for experiment 4.

The first scenario considers just two previous execution, second scenario considers four executions, and third scenario eight previous executions. Each scenario would run 20 executions. Each execution requests to run an appliation of size 200.

First Scenario

In this scenario, two previous executions were considered for the statistical analysis. This experiment lasted eight and half hours. Our analysis starts with *komolongma*. Gridjobs selected three probability functions Cauchy, Logistics and Weibull. Logistics was the more accurate p.f. but Cauchy (location $\approx$ 240000 and scale $\approx$ 10) was used to model the resources over 83% of the times. When Cauchy predicted a resource behavior the 55% of the times under estimated the resource behavior with an error of the 1.7%. Forty five percent over estimates the resource behavior with less that 1% of error.

rocks-52 shows a larger variety of p.f. than *komolongma*. Six p.f. were considered by Gridjobs Gamma, Normal, Log Normal, Logistic, Cauchy and Weibull. Weibull(shape $\approx$ [20,22], scale $\approx$ [176000, 182000]) was choosed over 71% of the times and was the more accurate p.f. However, over 38% of the times it over estimated with an error close to the 9%. Sixty two percet of the times, it under estimetated with 6% of error.

Gridjobs modeled *fsvc001* with four p.f. Weibull, Log-Normal, Logistic and Cauchy. Cauchy was used 53% of the times and Logistic 33%. The Cauchy parameters were location $\approx$ 114000 and scale between 0 and 400. The Logistic parameters were location $\approx$ 114000 and scale between 350 and 500. These functions were the more accurate p.f. Cauchy over 47% of the times over estimated the resource performance with a 3% of error and under estimated 53% of the times with 3% of error, too.

Second scenario

Gridjobs used two p.f. to model the komolongma performance: Cauchy and Gamma. Cauchy was employed over 95% of the times. Fifty eight percent of the times the performace was over estimated with an error of 0.8%. The under estimation occurred 42% of the times with an error close to 1.4%. The location parameter was close to 240000 and scale was close to 5.

rocks-52 was modeled with three p.f. Normal, Cauchy and Weibull. Weibull was selected 85% of the times. Forty five percent of the times the performace was over estimated with 5.6% of error. The under estimation occurred 55% of the time with 8.6% of error. This p.f. has parameters with the following values: shape between 15 and 20 and scale around of 178000.

Gridjobs modeled the *fsvc001* behavior with three p.f. Cauchy, Gramma and Logistic. Cauchy was selected 85% of the times. This Cauchy function with a location value equal to 112000 and scale equal to 40, over estimated 49% of the times with 2% of error and under estimated over 51% of the times with a 2.3% of error.

Third scenario

komolongma was modeled from three p.f. Gamma, Logistic and Cauchy. Logistic selected over 39% of the times, shows the more accurate p.f. In addition, the estimation error is around of 1.5% and 2.4%. Cauchy modeled *komolongma* over 52% of the times. The error percentage is around of 0.5% and 2.3%.

Gridjobs modeled *rocks-52* through Gamma(4%), Logistic(4%), Weibull(9.5%), Normal(14.28%) and Cauchy($\approx$66%). Logistic and then Cauchy, were the more accurate p.f. Estimations based

on Cauchy suffered an error between 1.2% and 3.9%. Similarly, Logistic exhibited an error between 0.8% and 1.5%.

Finally, *fsvc001* was modeled through five p.f. Exponential(1%), Normal(5%), Weibull(5%), Log-Normal(20%) and Cauchy(68%). Different from previous executions, the estimation error reached between eight and eleven percent for the Cauchy p.f. and between 12 and 36 percent for Log-Normal.

5.5.4 Different Ranking Schemes

Gridjobs implements at least four ranking schemes. The first ranking scheme considers the application performance and sorts the resources in such a way the fastest resource goes first and the slowest goes last. An experiment to compare this ranking scheme with the classical Round

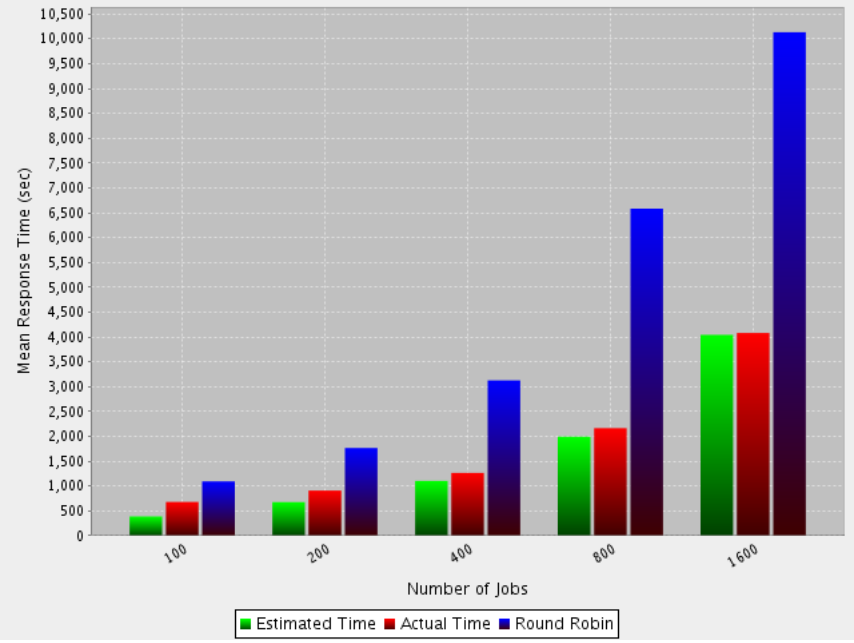

Figure 5.21. Behavior exhibited by the ranking scheme used in Gridjobs by default.

Robin Scheduler is depicted in Figure 5.21. In this experiment, Gridjobs with *defaultrank* scheme, run 25 executions. The executions were grouped by input data size. There were five groups, one group of size 100, other of 200, 400, 800 and 1600. From Figure 5.21 two observations can be made: the Gridjobs estimation improves while the input data size increases and *defaultrank* exhibits a performance to beat impressively the Round Robin performance.

The *expansion-factor* ranking scheme is also implemented in Gridjobs. This ranking scheme is based on the equation 5.1

$$\frac{execution\_time + waiting\_time}{execution\_time} \tag{5.1}$$

where $waiting\_time$ corresponds to the time that certain task spend enqueued. In particular, Gridjobs uses previous executions stored in GIS in order to do the corresponding estimations. Figure 5.22 shows a performance comparison between Gridjobs employing the *expansionfactor* scheme vs. the Round-Robin scheduler.

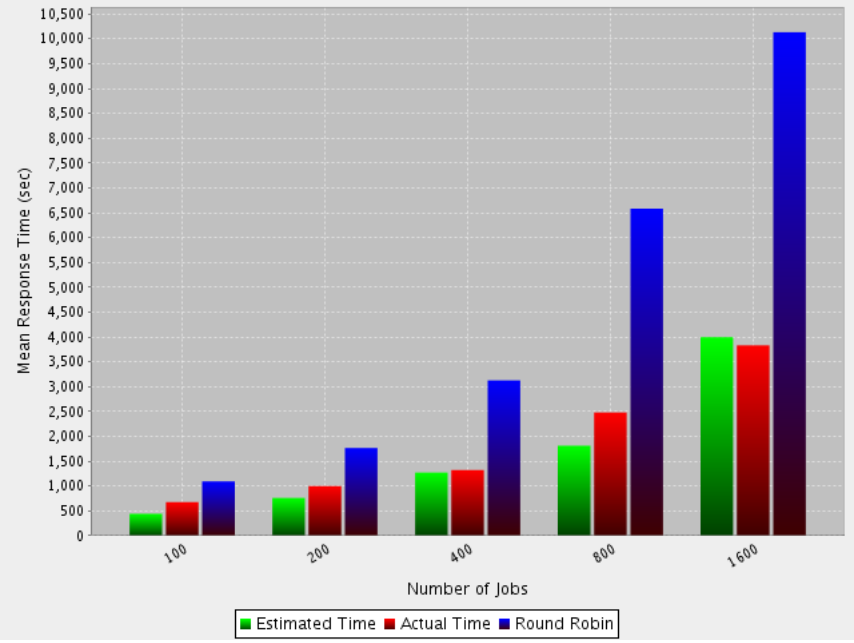

Figure 5.22. Behavior exhibited by the *expansionfactor* ranking scheme vs. the Round-Robin scheduler.

The *timeexecutionerror* ranking scheme has been also implemented in Gridjobs. This ranking scheme employs the expression 5.2 to sort the resources.

$$\frac{actualelapsedtime - estimatedelapsedtime}{estimatedelapsedtime} \tag{5.2}$$

Similarly to the previous ranking scheme, Gridjobs employs the information observed in previous executions and determines a value for each computational resource according to Equation

5.2. Figure 5.23 compares this scheme with the Round-Robin scheduler.

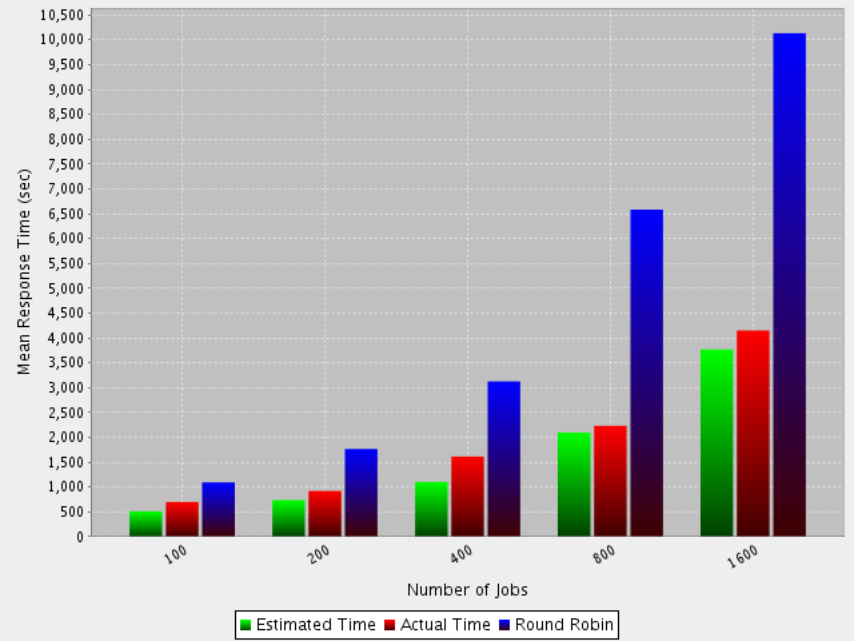

Figure 5.23. Behavior exhibited by the *timeexecutionerror* ranking scheme vs. the Round-Robin scheduler.

Finally, Gridjobs has also implemented a load-based ranking scheme. In this scheme Gridjobs queries the Grid Information Service(GIS) and retrieves the resource information to regard with the resource load. Gridjobs then goes through every computational resource, then reads the load per node in the aforementioned resource and averages the *fifteenload*. This average corresponds to the *fifteenload* of that cluster. *fifteenload* regards with the average number of enqueued tasks at CPU level during the last fifteen minutes. From that measure the resources are then sorted, the resource with lightest load goes first and heaviest goes last. Figure 5.24 shows a comparison between the load-based rank and *timeexecutionerror* (left subgraph) and the load-based rank and *expansionfactor* (right subgraph). On both graphs, the red bars correspond to the load-based rank. Although there is not a remarkable difference between the compared schemes, these preliminaries experiments suggest that ranking schemes to incorporate load metrics could beat those schemes who do not consider them.

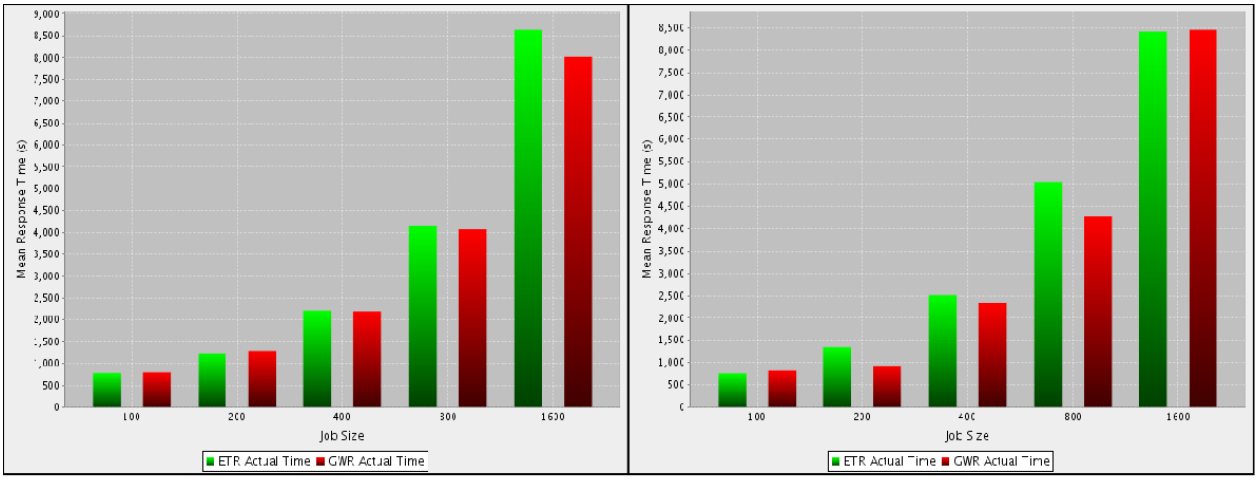

Figure 5.24. Behavior exhibited by the *timeexecutionerror* ranking scheme vs. the Round-Robin scheduler.

5.6 Summary

Gridjobs has been fully tested on computational resources to deploy GRAM services, Sun Grid Engine as LRM and SCMSWeb or Ganglia as monitoring tools. During our testing and experimental phases Gridjobs has achieved an important maturity degree. Since Gridjobs is a tool for real environments, its development has required a strong programming discipline to control and catch diverse abnormal circumstances. Gridjobs has also evolved in a computational gateway able to dispatch and track executions for over 48 hours of continuous executions. Although Gridjobs is a multi-thread application, it exhibits a significant stability and takes special care of avoiding concurrent access to shared resources.

During our experiments interesting results have been derived. First, resources barely can be modeled from one unique probability function. Gridjobs tries different probability functions to model the resource behavior and evaluates the different probability functions through the Kolmogorov-Smirnov test. The probability function to model a given resource vary over time and are highly sensible to changes observed in the environment.

Gridjobs provides a platform to predict performance on short periods. Previous works analyze larger periods of time and infer probability functions over those periods but barely consider suddendly changes in the infrastructure. Previous works have shown that Weibull is

an appropriate p.f. to model resource behavior. However, during our experiments Cauchy was selected most of the time and many times achieved levels of accuracy close to 99%. However, Logistic also exhibited important estimations. The Kolmogorov-Smirnov test integrated in Gridjobs, has shown an important degree of precission since most of the time it selected the more accurate p.f. Finally, Gridjobs evaluted different ranking schemes to consider failure rates, enqueued times, and resource performance.

CHAPTER 6

Conclusions and Future Work

The goal of this thesis has been to develop tools to understanding the resource management problem on large scale distributed system. To achieve this goal we have developed a framework, referred to as GridJobs, that integrates deployment, execution, and notification mechanisms for running applications across multiple diverse clusters. The main feature of this framework is the use of statistical analysis over historical data combined with self-adaptive mechanisms to automatically select efficient resources is a grid environment. GridJobs exhibits a modular design that allows the integration of new technologies and algorithms. Gridjobs was implemented assuming the existence of a computational environment to deploy a minimal set of grid compliant services and protocols. Gridjobs thus assumes the availability of GRAM and a monitoring tool, such as SCMSWeb or Ganglia. In addition, since Gridjobs is a platform to work with computational grids, it is expected that GRAM deployments be properly configured with some Local Resource Manager instance, such as SGE, PBS and LSC.

Grid environments present challenges not only at the user level but also at the administration level. To manage a grid resource demands time and effort. For instance, an operational grid resource requires software updates, add/remove users, update certificates, re-start services, manage disk space, add/remove compute nodes and many others duties. In particular, PRAGMA infrastructure where barely exists a central management authority, delegates the responsibility of keeping a healthy resource over each resource owner. PRAGMA embraces 30 clusters but many of them exhibit some kind of problem to avoid its full utilization. Gridjobs has been tested over diverse PRAGMA grid computational assets. Some of them deploy

out-dated and buggy management scripts to impose an additional level of uncertainty about resource performance. For instance, rocks-52's GRAM at San Diego Supercomputing Center could not return the output of a task which has been submitted with the globus-job-submit command. Similarly, fsvc001 at the AIST in Japan occasionally dropped tasks with no apparent reason. These factors induce additional noise over one already unsteady infrastructure. A very important contribution of GridJobs is that it takes care of abnormal circumstances. Gridjobs deploys a task module to track closely the behavior of tasks submitted by the framework. In that sense, any abnormal activity presented by a GRAM service or an unexpected long execution time are handled properly by the framework in order to avoid unnecessary resources consumption. Normal and abnormal circumstances are recorded and used by Gridjobs statistical module in order to model posterior resources behavior.

Previous works have assumed that one unique probability function could model the behavior of a given resource. Those works analyzes data collected two or three years ago and create macro models to represent the resources behavior. Despite important results have been derived, it is a misleading conclusion to claim that Weibull, for instance, models absolutely a resource. Dynamic systems present suddenly events to affect any static performance prediction.

It is hard to claim that a workload present in a given resource stays constant in time. Gridjobs integrates a simple procedure to predict with a significant accuracy degree the performance exhibited by a given resource. That information along with information provided by monitoring tools allows deriving accurate performance behaviors. However, this simple procedure requires some kind of stabilizer mechanism because it is highly reactive to small changes on the perceived performance.

Nowadays, Computational grids like PRAGMA, deploy Java based resource management services like GRAM. This mere fact suggests that it is an error to think computational grids in terms of High Performance Computing environments since Java is not characterized by its performance profile. However, there are additional reasons. First, each task executed under GRAM service creates a GRAM manager instance on the target resource. When multiple tasks

are submitted to single computational asset, the asset's head node starts to be overloaded because many gram managers are in memory (aka. Java processes). Second, GRAM simplifies the interaction with remote LRMs but imposes an overhead in the response time of the tasks submitted under it. For instance, when GRAM receives a task, it delivers this task to LRM. LRM in turn dispatches this task toward a free compute node meanwhile GRAM is waiting for a LRM notification. When LRM notifies, GRAM takes some time to realize about the notification. This time could last 5 seconds which is unacceptable for real time applications. Gridjobs divides the scheduling problem over unsteady and scattered computational resources in two problems: performance estimation and decision making. The performance estimation procedure is carried out from information provided by the statistical module. Then, Gridjobs sorts the computational assets according a ranking scheme given by the framework user. Finally, using a divisible load approach to meet the two aforementioned conditions, the load is divided amongst the available resources.

The work presented in this thesis can be extended in several directions. First, we plan to investigate the effect of implementing negotiation of multiple instances of GridJob. Implementing negotiation may lead to a more concerted utilization of the resources; however it raises issues related to the scalability of the resource selection algorithms. Second, we have assumed that security and authorization functionalities are provided by a middleware such as Globus. However, secure and trustiness considerations may change the behavior of the systems and the interaction among Gridjobs instances. Finally, more work is needed to incorporate other metrics considerations such as reliability, availability and sustainability (e.g. energy consumption and carbon emissions) of the system.

CHAPTER 7

Ethics

Computers and information systems have reached several aspects of our daily life. Issues associated with family, education, careers, freedom and democracy have been permeated by computational infrastructures to empower the development of more sophisticated tasks. In the mid 1940s, Wiener envisioned "a second industrial revolution, an 'automatic age' with 'enormous potential for good and for evil' that would generate a staggering number of new ethical challenges and opportunities." Wiener's books remarked the relevance of issues such as security, unemployment, responsabilities of computer professionals, religion, globalization, virtual communities, and teleworking, among others.

Computational resources would provide an environment to foster situations "to engage in creative and flexible actions and thereby maximize their full potential as intelligent, decision-making beings in charge of their own lives". From this perception of the human life purpose led Wiener to adopt the "great principles of justice". Those principles are principles of freedom, equality and benevolence.

In 1960s, those ethics principles changed toward one principle, the principle of openness. The software movement directed a proposal in which the decision making process and development of new software solutions were not determined by one unique person or corporation but many with altruist and common motivations. The ethic principles were intrinsically and implicitly defined by the community itself. Those community members who exhibit succeeded results are implicitly followed by others in the community. The collaborative movement started.

Many Internet-based projects to follow the aforementioned approach have reached an

amazing success such as Linux, Firefox, YouTube, Wikipedia, and Facebook, among others. Those projects are now driven by additional principles such as peering, sharing and acting globally. Many project's contributors are motivitaed because the acknowledgement given by the users community. There are not explicit rules, the community itself rewards or punishes the performance of its members. In addition, protocols and services employed by the global village provide a suitable environment with minimum set of operational conditions such as security and privacy.

Nowadays, the global village not only shares software but hardware resources. Collaborative hardware initiatives, aka. Grid, provide a great opportunity to small and mid size institutions to use sophisticated computational facilities. Large institutions are willing to share their massive and expensive computational infrastructures for free. Some of them are motivated because of their interest to leverage collaborative research amongst international peers. Tangible resources differ from software projects because formers have implicit costs such as maintenance and operation. Thus, "inhabitants" of the global village must carefully use the shared resources. Protocols and services in charge of that kind of resources would seek to create environments to favor user demands and observe resources policies.

Non-intrusive solutions to guarantee a fair utilization of the resources according to provider expectations would prevail. Gridjobs respects management and privacy policies established by the resource owner. It highly relies on mechanisms deployed on the resource in such a way that those mechanisms fulfill the requirements to assure the proper operation of the resource. In addition, because its non-invasive nature, it is able to co-exist with existing processes and adapts automatically its behavior according to resources availability.

Today, the ethic principles are more important than ever. Lack of responsability on duties given to everyone has created a dark landscape. Pandemies, weather changes, rapid resources depletion, and contamination, among others; are problems resulting of our greedy behavior. Unselfishness would characterize our actions at different levels of our life such as family, friends, community and work. Collaboration, cooperation and sensibility for our environment are required to reduce the negative impact of our predecessors.

APPENDICES

APPENDIX A

Code in Grails

Grails is the selected platform to develop Gridjobs. Grails is a project derived from Groovy which is a scripting language for the Java Virtual Machine (JVM). Hence, Grails and Groovy are programming environments to gracefully interoperate with existing techonlogies for the Java programming language. The power of Grails is that it leverages the rapid development of highly modular web applications. This modularity is easily inherited to applications developed under Grails.

Characteristics exhibited by Grails and Gridjobs modules code are described as follow.

A.1 Grails and Databases

Grails provides an abstraction layer named Grails object relational mapping (GORM). This layer is on top of Hibernate which is an object relational mapping(ORM) for the Java language. Hibernate provides a framework for mapping an object oriented domain model to conventional databases.

GORM accepts the domain class definition following the Groovy syntax and transforms into SQL sentences for building tables, indexes, and primary keys, among others; over relational databases to implement the JDBC API. Figure A.1 shows, on the left side, the definition of a *Gridresource* according to the Groovy syntax. On the right side, a PostgreqSQL table used to represent the *Gridresource* domain class is shown.

```
class Gridresource {
   static hasMany = [jobstates:Jobstate,
         task: Task,
         taskestimationtime: Taskestimationtime,
         application: Application]
   String headnode
   String name
   String batchscheduler
   String country
   String organization
   String userhome
   static constraints = {
      name(unique:true,nullable:false)
   }
}
```

```
                                    Table "public.gridresource"
     Column     |          Type          | Modifiers
----------------+------------------------+-----------
id              | bigint                 | not null
version         | bigint                 | not null
headnode        | character varying(255) | not null
name            | character varying(255) | not null
batchscheduler  | character varying(255) | not null
country         | character varying(255) | not null
organization    | character varying(255) | not null
userhome        | character varying(255) | not null
Indexes:
   "gridresource_pkey" PRIMARY KEY, btree (id)
   "gridresource_name_key" UNIQUE, btree (name)
```

Figure A.1. Definition of the *Gridresource* class and its corresponding representation on the PostgreSQL database.

A.2 Services in Grails

Grails follows a Model-View-Controller (MVC) architecture. The model is represented through relational dabases schemes, the view is implemented through diverse web based technologies such as JSP, Java Server Faces and AJAX and finally the controllers are implemented as services.

A Grails service is a class that holds one or more methods to implement the business logic. The definition of a `version` service is described as follows. In order to create a service, the `grails create-service version` command is executed, Figure A.2. Prior to enable the remote access, it is necessary to install the `remoting` plug-in. When it has been installed, the services might expose their functionality to external entities by adding the `static expose = []` sentence. This sentence instructs to Grails to reveal the service functionality through the protocols specified between the square brackets. For example, Figure A.2 shows how to include the `expose` sentence and how a service method is implemented. In this case, the `version` method is accessible through the *Hessian* protocol![1]

Although the methods are exposed, it is still necessary to implement a Java interface where signature of the methods are declared. This interface must be shared with every remote entity willing to access the service functionality. Figure A.3 presents a Java interface to implement the signature of the `version` method defined in Figure A.2.

[1] Hessian is a binary web service protocol implementation.

```
1  class VersionService {
2     static transactional = false
3     static expose = ['hessian']
4     String version() {
5        return "gridjobs 0.3.1"
6     }
7  }
```

Figure A.2. Service accessible through the Hessian web service protocol.

```
1  interface Version {
2     String version();
3  }
```

Figure A.3. Definition of *Version* interface.

Lastly, Figure A.4 shows the final implementation of the `VersionService` class.

```
1  class VersionService implements Version{
2     static transactional = false
3     static expose = ['hessian']
4     String version() {
5        return "gridjobs 0.3.1"
6     }
7  }
```

Figure A.4. *VersionService* fully implemented and ready to be accessed via Hessian protocol.

The `version` method could be used by either clients or other Gridjobs peers for querying the version of a Gridjobs instance and to determine if compatibility problems could arise. On the client side, the interface definition must be located in a directory accessible to the JVM client. This interface can be seen like a contract between client and server sides. Through this contract, the client side knows what methods are available and which parameters are required for a proper message interchange. It thus provides a basic communication primitives to leverage the communication between both sides. In particular, the `version` method when is invoked, it returns a `String` to represent the Gridjobs version.

A.3 Gridjobs modules in Grails

A.3.1 Deployment Module

The Deployment module defines a service which implements the `Deploy` interface shown in Figure A.5. The `deploy` method expects all textual information relevant to the application.

Figure A.6 shows the implementation of the `DeployService` module. First of all, the informa-

```
1  interface Deploy {
2     String deploy(String user, String host,
3       String appname, String executablename,
4       String appdir, String data, String basedir,
5       String script, String app);
6     boolean checkapp(String appname);
7     boolean removeapp(String appname);
8  }
```

Figure A.5. Interface to define a basic set of methods used for deploying applications.

tion associated with the application is stored in IS, Lines 12-17. (That information is used later by Gridjobs when a user requests to run the application.) Then, the script file containing the steps used for deploying the application along with the application files are sent from the client to the framework server through an upload service implemented in the *Gridjobs* framework. Next, the script is copied to the remote resource, Lines 19-21, as well the application code, Lines 23-25. The installation script then is executed, Lines 27-28 and the script file and the application code finally are deleted from the remote resource filesystem, Lines 30-34.

This module can be accessed by a client program able to interact with other servers through Hessian protocol. During initial tests we have implemented a GUI using the Groovy language for easy deployment of various Unix legacy applications on diverse PRAGMA computational resources, Figure A.7.

A.3.2 Task execution module

The task execution module is in charge of running application instances over remote grid resources. It employs information from different modules and builds executions to use Grid protocols for running legacy applications or system commands. Gridjobs allows that remote entities execute grid applications through web service protocols. Thus, the `remote.Globusjob` interface defines two methods for executing tasks to serve as a contract between Gridjobs and task invoker, Figure A.8. This module supports asynchronous and synchronous executions. Through the `sync` parameter the execution invoker indicates what sort of execution is required. When `sync` variable is `true` then synchronous execution is carried out, otherwise asynchronous execution.

On server side, `LaunchService` implements the aforementioned interface. `LaunchService` provides the most elementary mechanism for task execution and exposes its functionality to third

```
1
2  class DeployService implements remote.Deploy {
3
4     boolean transactional = false
5     static expose = ['hessian']
6
7     String deploy(String user, String host,
8       String appname, String executablename,
9       String appdir, String data, String basedir,
10      String script, String app) {
11        ...
12        def gr = Gridresource.findByName(host)
13        def application = new Application(gridresource: gr,
14          installationpath: appdir,
15          executablename: executablename,
16          installationdate: new DateTime().toDate(),
17          datapath: data, name: appname)
18        ...
19        def command = "scp ${basedir}/${script} " +
20                      "${user}@${host}:${destdir}"
21        tmpout = util.Util.executegetoutput(command)
22        ...
23        command =
24         "scp ${basedir}/${app} ${user}@${host}:${destdir}"
25        tmpout = util.Util.executegetoutput(command)
26        ...
27        command = "ssh ${user}@${host} source ${script}"
28        tmpout = util.Util.executegetoutput(command)
29        ...
30        command = "ssh ${user}@${host} rm ${script}"
31        tmpout = util.Util.executegetoutput(command)
32        ...
33        command = "ssh ${user}@${host} rm ${app}"
34        tmpout = util.Util.executegetoutput(command)
35        ...
36    }
37 }
```

Figure A.6. Snippet code of the *DeployService* service implementation.

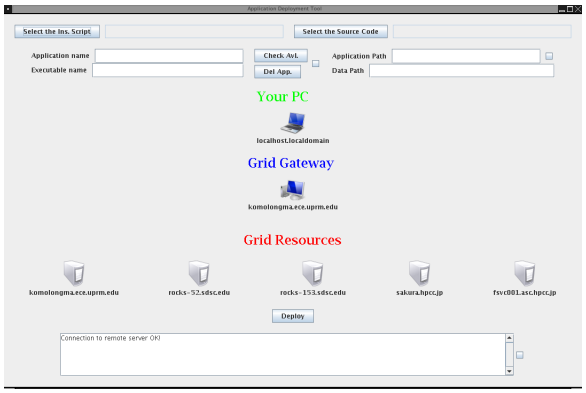

Figure A.7. Screen shot of the application used for deploying applications over PRAGMA clusters.

```
1  package remote
2
3  interface Globusjob {
4     String execute(String server,String jobmanager,
5                    String parameters, String when,
6                    boolean lock, boolean sync);
7     String execute(String server, String appname,
8                    String parameters,
9                    String executionname);
10 }
11
```

Figure A.8. Methods defined for invoking the task execution.

parties through the *Hessian* protocol. When asynchronous execution is requested, this service schedules a Quartz trigger that will attend this petition. The user can specify one or multiple executions of the same task because the `when` parameter of the `execute` method is mapped onto a `CronExpression`. For instance, the expression `"0 0 * MON,WED,FRI MAR,APR,MAY ? 2009"` indicates to execute the given task, every hour on Monday, Wednesday and Friday days during March, April and May months of 2009. However, this functionality offered by Quartz is barely used inside Gridjobs.

Figure A.9 shows the implementation of the `execute` methods defined in `remote.Globusjob` interface. The first `execute` method for tasks execution demands that the user provides every single detail required by Gridjobs for building the `globus-job-command` which allows the execution. For passing parameters between different instances of Quartz Jobs, Gridjobs employs a hash map defined in each Quartz Trigger. However, for simplicity the details related with trigger initialization have been omitted but this trigger stores the aforementioned parameters provided by the user along with additional parameters created by the framework in order to satisfy the grid middleware execution requirements. Then, all the collected information will be employed when the scheduled trigger be awoken, Line 15.

The second `execute` method provides a higher abstraction for executing tasks, Line 17-36. In this case, details about LRM, executable name, installation path and data path are retrieved from IS. However, when the information is collected, it invokes the `execute` method described above.

When the trigger is awoken, an instance of the `GlobusjobinstanceJob` class is created, Figure A.10. In line 6, all the information stored in the trigger is made accessible through a `HashMap` list.

```
class LaunchService
implements remote.Globusjob {
    boolean transactional = false
    static expose = ['hessian']
    def quartzScheduler

   String execute(String server, String jobmanager,
      String parameters, String when, boolean lock,
      boolean sync) {
      def cronexpression =
          util.quartz.Util.createcronexpression(when)
      Trigger trigger = new CronTrigger()
      // Initializing trigger parameters

      quartzScheduler.scheduleJob(trigger)
   }
   String execute(String server, String appname,
      String parameters, String executionname) {
      def gr = Gridresource.findByName(server)
      def batchscheduler = gr.batchscheduler
      def userhome = gr.userhome
      def criteria = Application.createCriteria()
      def results = criteria.list {
         and {
            like('name',appname)
            gridresource {
                  like('name',server)
            }
         }
      }
      def when = "NOW+2s"
      def lock = false
      def sync = false
      def newparameters =
        "remotecommand:${userhome}/" +
        "${installationpath}/${appname} " +
        "${parameters},function:${executionname}"
      execute(server, batchscheduler, newparameters,
              when, lock, sync)
   }
  }
```

Figure A.9. Snippet code of the *LaunchService*.

From the information contained in this list, the GlobusjobinstanceJob determines if the trigger will be fired in the future, the resource name, the LRM instance, and execution parameters, among others. Now, under asynchronous execution, the GRAM service informs about the different stages traversed by the executed task. Gridjobs fires periodically Java threads (instances of GlobusjobstatusJob) which surveil the task performance. However, a special care must be taken about the monitoring frequency because a relentless monitor can overwhelm a GRAM service or saturate a slow network connection. Gridjobs determines dynamically a suitable threshold for monitoring frequency, Lines 15-18. From the estimated time, a new trigger is created, line 30. This trigger will invoke the creation of a GlobusjobstatusJob instance which implements control mechanisms for recording changes on task status and taking away non-responding tasks. Finally, all the task information is purveyed to the trigger, Lines 31-38, and IS module, Lines 39-44; and the task is executed using the globus-job-submit command, Lines 48-52. This command returns an end point reference (EPR). This EPR is also stored in the recently created trigger and used onward for the GlobusjobstatusJob class in order to track the task performance.

When the trigger created by the GlobusjobinstanceJob is activated, it launches a new GlobusjobstatusJob instance, Figure A.11. This instance queries, on periodic intervals of time, the remote GRAM service through the globus-job-status command. This command returns the status of the task associated with the EPR provided by the globus-job-submit command invoked by the GlobusjobinstanceJob instance, Lines 13-16. The returned value is compared against the current task status, Line 18. If the status has not changed, this instance validates that the maximum time that this task might last in this state has not been exceeded, Lines 19-20. When the time is exceeded the monitor removes the task and reports the event to IS, Lines 21-29. This validation avoids to stay monitoring tasks whose performance has decreased or achieve an endless state due to abnormalities to happen in the remote resource or task perse. If the elapsed time in the current state has not been exceeded, a new trigger is programmed for validating the task status in a near future, Lines 31-45. When the status changes, this event is recorded and the Globusjobstatus instance validates if the new status is DONE or FAILED. On any case, the task has ended its execution therefore this event is notified to IS as well to

```
class GlobusjobinstanceJob{
   static triggers =  { }
   def quartzScheduler
   def execute(context) {
       ...
       def mjdm = context.getMergedJobDataMap()
       def server = mjdm.server
       def jobmanager = mjdm.jobmanager
       def parameters = mjdm.parameters
       def function = mjdm.function
       def crnexpr = mjdm.crnexpr
       def sync = mjdm.sync
       ...
       if (!sync) {
         def seconds =
           util.grid.Util.dryrunaveragetime(server,
                 jobmanager,
                 config.Config.iterations)/1000
         // Validation when seconds equals 0
         seconds =
           util.Util.maximum(
             config.Config.minimumthreshold,
             (int)(seconds + 1))
         def cronexpression =
           new CronExpression(
             util.quartz.Util.createcronexpression(
               "NOW+${seconds}s"
             )
           )
         def trigger = new CronTrigger()
         trigger.jobDataMap.server = server
         trigger.jobDataMap.jobmanager = jobmanager
         trigger.jobDataMap.function = function
         trigger.jobDataMap.parameters = parameters
         trigger.jobDataMap.crnexpr = crnexpr
         trigger.jobDataMap.sync = sync
         trigger.jobDataMap.cronexpression =
           cronexpression
         Gridresource _gr =
           Gridresource.findByName(server)
         Task _task = new Task(gridresource: _gr,
            unsubmitted: cdt,
            state: config.Config.UNSUBMITTED)
         _task.save()
         trigger.jobDataMap.status =
           config.Config.UNSUBMITTED
         trigger.jobDataMap.minimumthreshold = seconds
         def output =
           util.Util.executegetoutput(
             "${globushome}/bin/${globusjobsubmit} " +
             "${server}/jobmanager-${jobmanager} " +
             "${parameters}")
         trigger.jobDataMap.url = output
         if (!CronExpression.isValidExpression(crnexpr)){
            util.quartz.Util.removetrigger(context)
         }
         quartzScheduler.scheduleJob(trigger)
         return
       }
       ...
   }
}

```

Figure A.10. Snippet code of the *GlobusjobinstanceJob* class.

the user who submitted the task, Lines 47-57. Otherwise, if the new state is PENDING or ACTIVE; the time when the new state was reached, is saved in to IS and a new trigger is created for posterior monitoring of the task, lines 58-64.

This cycle is repetead until the task achieves a DONE or FAILED stage or the task exceeds a threshold of time that Gridjobs estimated.

A.3.3 Monitoring Module

The monitoring module is in charge to interact with either SCMSWeb or Ganglia monitoring tools via either HTTP protocol or plain TCP sockets, respectively. Gridjobs, in the production release, queries the remote monitoring tool each 5 minutes, hence, the `"0 0/5 * * * ?"` value is assigned to the `cronExpression` variable. Figure A.13 presents a snippet code of the `execute` method. Although the resources to be monitored appear hardcoded in Lines 3-7, the production release retrieves the resources information from IS. As follows, the interaction with a SCMSWeb instance is described. The remote resource is contacted through its web service via HTTP, Lines 9-20. The module expects a XML stream to contain general cluster information such as number of nodes, number of dead nodes, and cluster operating system; in addition, the XML file also provides a detailed summary of every compute, such as CPU, memory, storage, OS version, number of network interfaces, and network activity. This information is stored in a temporary file, Lines 21-27. The XML stream containing general cluster information is returned and stored in a temporary file, Lines 21-27. For Ganglia systems, Gridjobs uses plain sockets, Figure A.14. For getting information associated with the status of a remote resource, Gridjobs establishes a channel communication with the aforementioned resource through a socket connection. For that, it employs the resource name along with the port number. By default, Ganglia listens for connections at port number 8649. Then, the information coming throgh the socket is stored in a local file. In either way, the new file is then parsed and relevant information is stored in IS.

The retrieved information allows to resemble a rough view of the grid resources status. Thus, other services such as statistical module and service schedulers would benefit of this info in order to build execution plans to accurately predict the resource and task behavior.

```
class GlobusjobstatusJob
{
   static triggers = { }
   ...
   def quartzScheduler
   def mailService
   def execute(context) {
      def _flag
      def mjdm = context.getMergedJobDataMap()
      def oldtrigger = context.getTrigger()
      def previousstatus = mjdm.status
      ...
      def status = util.Util.executegetoutput(
        "${globushome}/bin/" +
        "${config.Config.globusjobstatus} " +
        "${url}")
      ...
      if (previousstatus == status) {
         if ( (maxwaitingtime + mjdm."${status}") <
              new DateTime().getMillis() ) {
            ...
            util.quartz.Util.removetrigger(context)
            ...
            _task =
               Task.findByUnsubmitted(mjdm.UNSUBMITTED)
            _task.exitstatus = config.Config.FAILED
            _task.output = "Time exhausted"
            ...
         }
         ...
         def trigger = new CronTrigger()
         // Put information required for next iteration
         trigger.jobDataMap.server = mjdm.server
         trigger.jobDataMap.jobmanager =
           mjdm.jobmanager
         trigger.jobDataMap.function = mjdm.function
         trigger.jobDataMap.parameters =
           mjdm.parameters
         trigger.jobDataMap.status = status
         ...
         scheduleJobWithOldTriggerName(context,
           trigger)
         ...
         return
      }
      ...
      if (status == config.Config.DONE ||
          status == config.Config.FAILED) {
         ...
         def contacts = Contactinfo.list()
         contacts.each { contact ->
            // send a notification mail
         }
         ...
         _task.save()
         ...
      } else {
         def trigger = new CronTrigger()
         // Put information required for next iteration
         ...
         scheduleJobWithOldTriggerName(context,
           trigger)
         ...
      }
      ...
   }
   ...
}
```

Figure A.11. Snippet code of the *GlobusjobstatusJob* class.

```
class StatisticsJob {
   def cronExpression = "15 0/20 * 1 12 ?"
   def group = "mygroup"
   def execute() {
   }
}
```

Figure A.12. Snippet code of the *StatisticsJob.groovy* file.

```
public execute() {
  ...
  def listserver = [
    "komolongma.ece.uprm.edu",
    "fsvc001.asc.hpcc.jp",
    "sakura.hpcc.jp",
    "rocks-152.sdsc.edu"
  listserver.each { server ->
    def urlsuffix =
      "cgi-bin/scmwsweb/xml_display.cgi?grid=on"
    def u =
      new URL("http://${server}/${urlsuffix}")
    def uc = u.openConnection()
    def conn = (HttpURLConnection) uc
    conn.setDoOutput(true)
    conn.setDoInput(true)
    conn.setRequestMethod("GET")
    def bufferInput = new BufferedReader(
       new InputStreamReader(
         conn.getInputStream()))
    def time = new Date().time
    def outFile = new File("${server}-${time}")
    def string
    while ((string = bufferInput.readLine()) != null)
    {
       outFile.append(string + "\n")
    }
    ...
    // Saving to database
    def xmlparser =
      new XmlParser().parse(outFile)
    def cluster = new Cluster(xmlparser.cluster[0])
    cluster.save()
  }
}
```

Figure A.13. Snippet code of the monitoring module.

```
def socket = new Socket(resourcename, new Integer(monitoringport))
def input = new DataInputStream(socket.getInputStream())
def line = null
while ( (line = input.readLine()) != null) {
   outFile.append(line + "\n")
}
input.close()
socket.close()
```

Figure A.14. Snippet code to show the interaction with a Ganglia monitoring tool.

A.3.4 Statistical Module

The statistical module uses information observed by the monitoring module and finds probability functions to approximately represent the resources behavior. This module could be invoked in a synchronous or asynchronous way, however a description of the synchronous part is given as follows. During our experimental phase, this module was invoked every hour. The `"0 0 0/1 * * ?"` string assigned to `cronExpression` indicates that this module will be executed every hour. When the statistical module is fired, it interacts with the R statistical tool using R scripts generated dynamically by Gridjobs. These scripts are in charge of finding probabilistic functions to best represent the data recorded by the task module. Most of the data used for this analysis were collected by the task module. All events registered during the execution of a task are saved in log files as well as database records. Each time that a task either ends its execution or achieves a failure state, the framework records the different times registered by the task during its execution. For instance, Figure A.15 shows the content of a Gridjobs log file which has achieved a successful termination.

The statistical modules invokes the `processrawdata.groovy` script (Figure A.16) which processes a large bunch of log files and retrieves the values associated with the UNSUBMITTED, PENDING, ACTIVE and DONE fields. The numbers in front of the aforementioned labels are timestamps to represent moments in the time when the task changed of stage. Now, the elapsed time of a task in a given stage is the result of substract the aforementioned stage from the next stage. For instance, the elapsed time in UNSUBMITTED stage of the task represented in Figure A.15 is calculated as follows. The value of the UNSUBMITTED field is 1219989644716 (2:00:44 - 29/08/2008) and 1219989648014 (2:00:48 - 29/08/2008) for the PENDING field. To substracting UNSUBMITTED from PENDING is 3298. It means that this task lasted in UNSUBMITTED state more than 3 seconds. Thus, the values associated to the fields UNSUBMITTED, PENDING, ACTIVE and DONE are extracted and saved in a list for beign processed later.

`processrawdata.groovy` also generates a set of statistical graphics such as density plots, run sequence plots, autocorrelation function plots, histogram and quantile plots. The graphics

```
DONE
server: komolongma.ece.uprm.edu
jobmanager: sge
function: experiment-1c
parameters: ...
UNSUBMITTED: 1219989644716
PENDING: 1219989648014
ACTIVE: 1219989650077
getoutputtime: 1219989761237
DONE: 1219989758022
```

Figure A.15. Log file containing a task execution summary.

```
1  public execute() {
2    ...
3    def binding = new Binding()
4    def gs = new GroovyScriptEngine(".")
5    def script = "${scripthome}/processrawdata.groovy"
6    binding.setVariable("starttime",starttime)
7    binding.setVariable("endtime",endtime)
8    binding.setVariable("servers",_servers)
9    binding.setVariable("statuses",_statuses)
10   gs.run(script,binding)
11 }
```

Figure A.16. Snippet code used for processing the log files generated by the task monitor module.

are generated per each resource and stage using R graphical functions, Figure A.17. The R script used for creating the aforementioned graphics is generated on-fly in order to capture the unsteady essence of the grid infrastructures. The graphics generated are not used by the framework per se but they can be used for the framework user in order to get a visual support of the events happening with the grid resources.

```
1     Rfilename = "${scripthome}/fscript.R"
2     binding.setVariable("filename",Rfilename)
3     script = "${scripthome}/fscriptR.groovy"
4     gs.run(script,binding)
5     util.Util.executegetoutput(Rfilename)
```

Figure A.17. Snippet code to generate the statistical graphics associated with task performance.

REFERENCES

[1] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998. ISBN 1558604758.

[2] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002. URL `http://www.gridtoday.com/02/0722/100136.html`.

[3] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8105`.

[4] Heinz Stockinger. Defining the grid: a snapshot on the current view. *J. Supercomput.*, 42(1):3–17, October 2007. ISSN 0920-8542. doi: 10.1007/s11227-006-0037-9. URL `http://dx.doi.org/10.1007/s11227-006-0037-9`.

[5] Parvin Asadzadeh, Rajkumar Buyya, Chun Ling Kei, Deepa Nayar, and Srikumar Venugopal. Global grids and software toolkits: A study of four grid middleware technologies. *CoRR*, cs.DC/0407001, 2004.

[6] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2003.1195409.

[7] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. pages 104–111, 1988. doi: 10.1109/DCS.1988.12507. URL `http://dx.doi.org/10.1109/DCS.1988.12507`.

[8] Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.

[9] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[10] Karoly Bosa, Wolfgang Schreiner, Michael Buchberger, and Thomas Kaltofen. A Grid-Based Medical Decision Support System for the Diagnosis and Treatment of Strabismus, September 2006.

[11] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999. ISSN 0360-0300.

[12] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[13] Xiangliang Zhang, M. Sebag, and C. Germain. Toward behavioral modeling of a grid system: Mining the logging and bookkeeping files. pages 581–588, Oct. 2007. doi: 10.1109/ICDMW.2007.52.

[14] T. Kielmann. Programming models for grid applications and systems: Requirements and approaches. pages 27–32, Oct. 2006. doi: 10.1109/JVA.2006.41.

[15] Kaizar Amin, Mihael Hategan, Gregor Von Laszewski, and Nestor J. Zaluzec. Abstracting the grid. In *In Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP 2004), A Coruna*, pages 250–257, 2004.

[16] Zsolt N. Németh and Vaidy Sunderam. A formal framework for defining grid systems. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 202, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1582-7.

[17] Zsolt Németh and Vaidy S. Sunderam. Characterizing grids: Attributes, definitions, and formalisms. *J. Grid Comput.*, 1(1):9–23, 2003.

[18] Thomas G. Robertazzi. Ten reasons to use divisible load theory. *Computer*, 36(5):63–68, 2003. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2003.1198238.

[19] Olivier Beaumont, Arnaud Legrand, and Yang Yang. Scheduling divisible loads on star and tree networks: Results and open problems. *IEEE Trans. Parallel Distrib. Syst.*, 16 (3):207–218, 2005. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2005.35. Member-Henri Casanova and Senior Member-Yves Robert.

[20] Sivakumar Viswanathan, Bharadwaj Veeravalli, and Thomas G. Robertazzi. Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(10):1450–1461, 2007. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2007.1073.

[21] Yudith Cardinale and Henri Casanova. An evaluation of job scheduling strategies for divisible loads on grid platforms. In *in Proceedings of the High Performance Computing & Simulation Conference (HPC&S'06), Bonn, Germany.*, 2006.

[22] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems. *Software Practice and Experience*, 32:135–164, 2002.

[23] Rajkumar Buyya, David Abramson, and Jonathan Giddy. An economy driven resource management architecture for global computational power grids, 2000.

[24] Kevin Lai, Bernardo A. Huberman, and Leslie Fine. Tycoon: A distributed market-based resource allocation system, Apr 2004. URL `http://arxiv.org/abs/cs.DC/0404013`.

[25] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *ACM SIGOPS Operating Systems Review*, 43:16–25, January 2009.

[26] Ritu Sabharwal. Grid infrastructure deployment using smartfrog technology. In *ICNS '06: Proceedings of the International conference on Networking and Services*, page 73, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2622-5. doi: http://dx.doi.org/10.1109/ICNS.2006.54.

[27] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 9781595937643. doi: http://dx.doi.org/10.1145/1362622.1362680. URL `http://dx.doi.org/10.1145/1362622.1362680`.

[28] Eduardo Huedo, Rubén S. Montero, and Ignacio M. Llorente. A modular meta-scheduling architecture for interfacing with pre-ws and ws grid resource management services. *Future Gener. Comput. Syst.*, 23(2):252–261, 2007. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2006.07.013.

[29] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A framework for adaptive execution in grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.584.

[30] Chris Richardson. Orm in dynamic languages. *Queue*, 6(3):28–37, 2008. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1394127.1394140.

[31] Kurt Hornik. The R FAQ, 2008. URL `http://CRAN.R-project.org/doc/FAQ/R-FAQ.html`. ISBN 3-900051-08-9.

[32] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007. ISBN 1932394842.

[33] Graeme Rocher. *The Definitive Guide to Grails (Definitive Guide)*. Apress, Berkely, CA, USA, 2006. ISBN 1590597583.

[34] James Elliott, Tim O'Brien, and Ryan Fowler. *Harnessing hibernate*. O'Reilly, 2008. ISBN 9780596517724.

[35] Jennifer M. Schopf, Laura Pearlman, Neill Miller, Carl Kesselman, Ian Foster, Mike D'Arcy, and Ann Chervenak. Monitoring the grid with the globus toolkit mds4. *J. Phys.: Conf. Ser.*, 46(1):521+, 2006. ISSN 1742-6596. doi: 10.1088/1742-6596/46/1/072. URL `http://dx.doi.org/10.1088/1742-6596/46/1/072`.

[36] Jennifer Schopf, Mike D'arcy, Neil Miller, Laura Pearlman, Ian Foster, and Carl Kesselman. Monitoring and discovery in a web services framework: Functionality and performance of globus toolkit mds4. Technical report, Argonne National Laboratory, 2006.

[37] Philip Rizk, Cameron Kiddle, and Rob Simmonds. Improving gridftp performance with split tcp connections. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 263–270, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2448-6. doi: http://dx.doi.org/10.1109/E-SCIENCE.2005.53.

[38] Ernest Sithole, Gerard P. Parr, Sally I. McClean, and P. Dini. Evaluating global optimisation for data grids using replica location services. In *ICNS '06: Proceedings of the International conference on Networking and Services*, page 74, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2622-5. doi: http://dx.doi.org/10.1109/ICNS.2006.47.

[39] Cindy Zheng, David Abramson, Peter Arzberger, Shahaan Ayyub, Colin Enticott, Slavisa Garic, Mason J. Katz, Jae-Hyuck Kwak, Bu Sung Lee, Phil M. Papadopoulos, Sugree Phatanapherom, Somsak Sriprayoonsakul, Yoshio Tanaka, Yusuke Tanimura, Osamu Tatebe, and Putchong Uthayopas. The pragma testbed - building a multi-application international grid. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, page 57, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2585-7.

[40] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141–154, Feb 1988. ISSN 0098-5589. doi: 10.1109/32.4634.

[41] H.G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *Computers and Digital Techniques, IEE Proceedings* -, 141(1):1–10, Jan 1994. ISSN 1350-2387.