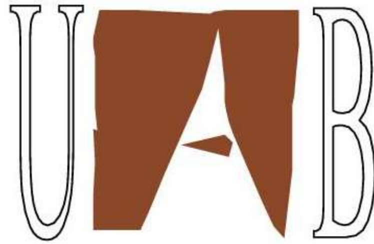


Universitat Autònoma de Barcelona

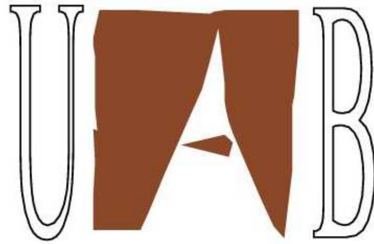
Escola Tècnica Superior d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Scheduling for Interactive and Parallel Applications on Grids

Memoria presentada por Enol Fernández del Castillo para optar al grado de Doctor por la Universidad Autónoma de Barcelona dentro del Programa de Doctorado en Informática, opción A: "Arquitectura de Computadores y Procesamiento Paralelo", bajo la dirección del Dr. Miquel Àngel Senar Rosell y la Dra. Elisa Heymann Pignolo.

Barcelona, julio de 2008



Universitat Autònoma de Barcelona
Escola Tècnica Superior d'Enginyeria
Departament d'Arquitectura de Computadors i Sistemes Operatius

Scheduling for Interactive and Parallel Applications on Grids

Memoria presentada por Enol Fernández del Castillo para optar al grado de Doctor por la Universidad Autónoma de Barcelona dentro del Programa de Doctorado en Informática, opción A: "Arquitectura de Computadores y Procesamiento Paralelo", bajo la dirección del Dr. Miquel Àngel Senar Rosell y la Dra. Elisa Heymann Pignolo.

Barcelona, julio de 2008

Abstract

Grid computing constitutes one of the most promising fields in computer systems. The next generation of scientific applications can profit from a large-scale, multi-organizational infrastructure that offers more computing power than one institution alone is able to afford. Grids need high-level schedulers that can be used to manage the resources spanning different organizations. These Grid Resource Management Systems (GRMS) have to make scheduling decisions without actually owning the grid resources, or having full control over the jobs that are running there. This introduces new challenges in the scheduling process done by the GRMSs. Although grids consist of many resources, and jobs submitted to grid may benefit from using them in a coordinated way, most of the Grid Resource Management Systems have focused on the execution of sequential jobs, with the grid being a large multi-site environment where jobs run in a batch-like way.

However, in this work we concentrate on a kind of jobs that have received little attention to date: interactive and parallel jobs. Interactive jobs require the possibility of starting in the immediate future and need mechanisms to establish a communication channel with the user. Parallel applications introduce the need for co-allocation, guaranteeing the simultaneous availability of the resources when they are accessed by the applications. We address the challenges of executing such jobs with a new architecture for a GRMS and an implementation of that architecture called the Cross-Broker. Our architecture includes mechanisms to allow the co-allocation of parallel jobs and the interaction of users with running applications. Additionally with the introduction of a multi-programming mechanism, a fast startup of jobs even in high occupancy scenarios is provided.

Agradecimientos

En primer lugar quiero expresar mi agradecimiento a Miquel Àngel Senar y Elisa Heymann por darme la oportunidad de realizar este trabajo y su dedicación a la dirección del mismo. A todo el Departamento de Arquitectura de Computadores y Sistemas Operativos y en particular a todos los compañeros de doctorado y a los integrantes del grupo de grid del departamento.

I would like to thank all the people of the CrossGrid and int.eu.grid projects. Thanks especially to Álvaro for all the help at the beginning and to Gonçalo, Herbert, Marcin, Marcus and Sven for the great moments I have shared with them. I would also like to thank Rainer Keller for hosting me at HLRS and for his support during my stay there. Thanks to Karen Miller for the English corrections.

Fuera del mundo del grid, me gustaría agradecer a mi familia y amigos por haberme apoyado desde el principio, aunque más de uno siga sin entenderlo. Gracias a Ana, por estar en todo momento compartiendo el camino conmigo.

Contents

List of Figures **xiii**

List of Tables **xv**

1 Overview **1**

2 Introduction **5**

2.1 Grids 6

2.1.1 Grid Architecture 7

2.1.2 Grid Middleware 8

2.2 Grid Infrastructure Initiatives 12

2.2.1 EGEE 13

2.2.2 OSG 15

2.2.3 DEISA 15

2.2.4 CrossGrid and Interactive European Grid 16

2.3 Grid Scheduling 20

2.3.1 Grid scheduling systems 21

2.4 Execution of Parallel and Interactive jobs on Grids 24

2.4.1 Parallel Jobs 25

2.4.2 Interactivity in Grid Environments 27

2.5 Contributions 28

2.6 Conclusions 28

3 An Architecture for Parallel and Interactive Jobs **31**

3.1 The Grid Environment 31

3.2 The Job Model 33

3.2.1 Job Lifecycle 35

3.2.2 Job Starters and Application Launchers 35

3.2.3 Interactive Agents . . . 37
3.3 Job Description Language . . . 38
3.3.1 Extended JDL . . . 40
3.4 CrossBroker Grid Scheduler . . . 46
3.4.1 A mechanism for multi-programming . . . 47
3.5 Conclusions . . . 50

4 CrossBroker Design and Implementation 53
4.1 Scheduling Agent . . . 54
4.1.1 User Access Module . . . 55
4.1.2 Scheduler . . . 57
4.1.3 Glidein Monitor . . . 58
4.2 Resource Searcher . . . 60
4.2.1 Resource Cache . . . 61
4.2.2 Matchmaking . . . 63
4.3 Job Execution . . . 65
4.3.1 Application Launcher . . . 65
4.3.2 Job Starter . . . 69
4.3.3 Interactive Agents . . . 72
4.4 Example Applications . . . 76
4.5 Conclusions . . . 81

5 CrossBroker Experimental Evaluation 83
5.1 CrossBroker Overhead . . . 83
5.1.1 Job Preprocessing . . . 84
5.1.2 Selection of Resources . . . 86
5.1.3 Remote Job Submission . . . 87
5.1.4 Job Start up and Execution . . . 88
5.1.5 Overall Overhead . . . 92
5.2 The CrossBroker on a real testbed . . . 92
5.2.1 The int.eu.grid testbed . . . 93
5.2.2 CrossBroker Usage . . . 94
5.3 Evaluation of the CrossBroker Mechanisms . . . 96
5.3.1 Workload Modelling . . . 96
5.3.2 Simulation of grid environments . . . 98
5.3.3 Co-allocation and Parallel Jobs . . . 99
5.3.4 Glidein and Interactive Jobs . . . 105
5.4 Conclusions . . . 110

6 Conclusions and Future Research 113
6.1 Open Lines of Research . . . 115

Bibliography 117

List of Figures

2.1 Layered grid architecture. 7
2.2 Hierarchical MDS structure 10
2.3 Globus components interaction. 10
2.4 UNICORE architecture. 12
2.5 The gLite core and local services. 14
2.6 The CrossGrid Testbed sites 18
2.7 The i2g infrastructure map. 19
2.8 Pacx Communication Structure 26

3.1 Grid environment architecture 32
3.2 Parallel job types 33
3.3 Job Lifecycle. 35
3.4 Job Starters and Application Launchers 37
3.5 Interactive Shadow and Interactive Agent 37
3.6 Interactive Job Execution 38
3.7 JDL job description. 39
3.8 Parallel JDL job description. 41
3.9 SubJobs specification in JDL. 42
3.10 Interactive JDL job description. 43
3.11 Example DAG. 44
3.12 JDL Dag description. 45
3.13 CrossBroker Architecture 47
3.14 Multi-programmed execution of jobs. 48
3.15 Multi-programmed execution of inter-cluster jobs. 50

4.1 CrossBroker Architecture 54
4.2 Structure of the Scheduling Agent 55
4.3 Use of GCB with glide-in 61

4.4 Structure of the Resource Searcher. 61
4.5 Job Execution Components. 66
4.6 MPICH-G2 execution on multiple sites 68
4.7 PACX-MPI execution on multiple sites 69
4.8 Interactive Agents in CrossBroker 74
4.9 Condor Bypass Interactive Agent 75
4.10 Plasma visualization application running on the Migrating Desktop . . 77
4.11 JDL file for the plasma visualization application 78
4.12 JDL file for the ANN application 79
4.13 Results of matchmaking 80

5.1 Average Overhead for Job Preprocessing 85
5.2 Average Selection Overhead 87
5.3 Average Remote Job Submission Overhead. 88
5.4 Average Glidein Start up Overhead 89
5.5 Average Execution Time of eIMRT 91
5.6 Runtime of eIMRT on virtual slots with different priority 91
5.7 Jobs submitted to the int.eu.grid infrastructure. 95
5.8 Jobs sizes in the int.eu.grid infrastructure. 95
5.9 Job arrival pattern in the workload 100
5.10 System Utilization for Worst Fit Policy 101
5.11 System Utilization for Best Fit Policy 102
5.12 System Utilization for First Fit Policy 103
5.13 Impact on the run time of co-allocated jobs 104
5.14 Cyclic job arrival pattern in the workloads 105
5.15 Distribution of runtimes 106
5.16 Average count of virtual slots. 109

List of Tables

3.1 Job structural and functional types 34

4.1 Errors Detected by the AL 70
4.2 Job Starter environment variables. 71
4.3 MPI-Start environment variables. 73

5.1 Production sites properties 93
5.2 Development sites properties 93
5.3 Number of jobs processed by CrossBroker 96
5.4 Results for low-load Workload 107
5.5 Results for high-load Workload 108
5.6 Metrics for interactive jobs. 109
5.7 Impact of Glidein on batch jobs. 110

CHAPTER 1

Overview

Grid computing constitutes one of the most promising fields in computer systems. The ability to use a large-scale, multi-organizational infrastructure opens new possibilities for the next generation of scientific applications that require more computing power than one institution alone is able to afford. A grid computing paradigm unites geographically-distributed and heterogeneous computing, storage, and network resources and provides unified, secure, and pervasive access to their combined capabilities.

Over the last years, several of these grid initiatives have created a computing infrastructure that provides ubiquitous and inexpensive access to a large amount of computational resources. Along with the hardware growth, the software for orchestrating the collaborative access to those resources has become more sophisticated and robust. This *middleware* hides the underlying physical infrastructure from the users, thus providing a unified and coherent vision of the grid environment, while maintaining site autonomy.

Resource management is an important subject for grids. It can be defined as the process of identifying requirements, matching resources to applications, allocating those resources, and scheduling and monitoring grid resources over time in order

to run grid applications as efficiently as possible. Grids consist of many resources, and jobs submitted to a grid may benefit from using them in a coordinated way. However, most of the Grid Resource Management Systems (GRMS) have focused on the execution of sequential jobs, the grid being a large multi-site environment where jobs run in a batch-like way.

There are other kinds of applications that have received little attention to date: interactive and parallel jobs. On the one hand, interactive applications allow the users to steer their application and to control them while running. These applications require the possibility of starting in the immediate future, while taking into account scenarios in which most computing resources might be running batch jobs. The grid scheduler should offer such services in an environment where it does not have complete control over the resources.

On the other hand, parallel applications can efficiently use many processors, taking advantage of the multiple resources in a grid system. Support for such applications introduces the need for co-allocation, i.e. the simultaneous or coordinated access of single applications of multiple types in multiple locations, managed by different autonomous resource management systems where reservation mechanisms may not be available. Moreover, even with the existence of the middleware layer that provides unified vision of the environment, the grid scheduler must deal with the different ways of starting and running applications, taking into account the Local Resource Manager System (LRMS) available at each resource and the different parallel application implementations.

In order to overcome these new challenges, in this work we present a novel architecture for the execution of interactive and parallel jobs in grid environments. This architecture includes the specification and definition of such jobs in a modular way, hence it allows the orthogonal combination of an interactive steering mechanism and parallel library implementations. We have also included a multi-programming mechanism that leverages the lack of control over the resources enabling the fast start up of jobs even in high occupancy scenarios and the co-allocation of applications without reservation support.

We have created an implementation of the proposed architecture called CrossBroker. CrossBroker, when users submit their application, makes the appropriate decisions and actions to run jobs on the remote resources without additional intervention. The system has been used as the main Grid Resource Management System in production environments in the European CrossGrid and the Interactive European Grid projects. CrossGrid evolved from 2002 to 2005 with the aim of extending the use of interactive and parallel applications in grids. Within the Interactive European Grid project that went from 2006 to 2008, the CrossBroker has been further refined and tested in a real environment.

The remainder of this thesis is organized as follows: Chapter 2 presents a general

introduction to grid environments and grid scheduling. Some grid initiatives are outlined and a review of the relevant related work in the topic is given. Chapter 3 proposes a novel architecture to support parallel and interactive jobs in these environments. The CrossBroker implementation details are given in Chapter 4, followed by the experimental validation in Chapter 5. Finally, conclusions and suggestions for future research are given in Chapter 6.

CHAPTER 2

Introduction

The next generation of scientific applications will require more computing power and storage than one single institution alone is able to afford. Grid environments constitute one of the most promising computing infrastructures for such applications. With the promise of high computational power at low cost, grid computing has become increasingly widespread around the world. The grid computing paradigm enables the sharing, selection, and aggregation of services of heterogeneous resources distributed across multiple administrative domains and provides unified, secure, and pervasive access to their combined capabilities. Grid computing, therefore, leads to the creation of virtual organizations by allowing geographically-distributed communities to pool resources in order to achieve common objectives.

Grids are becoming almost commonplace today, with many projects using them for production runs. The initial challenges of grid computing — how to run a job, how to transfer large files, how to manage multiple user accounts on different systems — have been resolved to the first order, such that users and researchers can now address the issues that will allow more efficient use of the resources.

In Section 2.1 we introduce most important grid concepts. Following this, Section 2.2 presents some of the latest grid projects and initiatives. In Section 2.3, we present

the challenges of grid scheduling and review related work in the area. In Section 2.4 we give an overview of the current status of the execution of parallel and interactive jobs on grids. Finally, we will review the main contributions of this thesis in Section 2.5.

2.1 Grids

Inspired by the electrical power grid's pervasiveness, ease of use, and reliability, in the mid-1990s the term "grid computing" [1] was proposed for an analogous infrastructure of wide-area parallel and distributed computing. A grid enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources, and specialized devices owned by different organizations for solving large-scale resource intensive problems in science, engineering, and commerce. Generally, for a system to be considered as a grid, it must meet the following criteria [2]:

1. A grid coordinates resources that are not subject to centralized control and at the same time addresses the issues of security, policy, payment, membership, and so forth that arise in these settings.
2. A grid must use standard, open, general-purpose protocols and interfaces. These protocols address fundamental issues such as authentication, authorization, resource discovery, and resource access.
3. A grid delivers nontrivial quality service, i.e. it is able to meet complex user demands (e.g. response time, throughput, availability, security, etc.).

The development of the grid infrastructure has become the focus of a large community of researchers. The grid systems need to solve several challenges originating from inherent features of the grid:

- Multiple administrative domains and autonomy. Grid resources are geographically distributed across multiple administrative domains and owned by different organizations. The autonomy of resource owners needs to be honored along with their local resource management and usage policies.
- Heterogeneity. A grid involves a multiplicity of resources that are heterogeneous in nature and will encompass a vast range of technologies.
- Scalability. A grid might grow from a few integrated resources to millions. This raises the problem of potential performance degradation as the size of the grid increases. Consequently, applications that require a large number of

geographically located resources must be designed to be latency and bandwidth tolerant.

- Dynamicity or adaptability. In a grid, resource failure is the rule rather than the exception. In fact, with so many resources in a grid, the probability of some resource failing is high. Resource managers or applications must tailor their behavior dynamically to use the available resources and services efficiently and effectively.

The grid goes further than simply sharing resources and data. A grid enables new scientific collaboration methods, termed e-Science [3], that tackle large scale scientific problems. e-Science enables massively distributed computation, the sharing of huge data sets almost immediately, and cooperative scientific work to gather new results.

2.1.1 Grid Architecture

Typically, grid architectures are arranged into layers [4] [5], where each layer builds on the services offered by the lower layer, in addition to interacting and co-operating with components at the same level.

Figure 2.1 shows the architecture stack of a grid proposed in [4]. It consists of five layers: fabric, connectivity, resource, collective, and application.

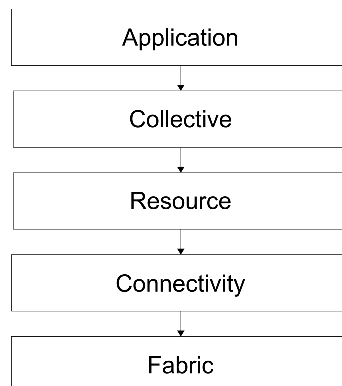


Figure 2.1: Layered grid architecture.

Fabric layer. The fabric layer defines the interface to local resources, which may be shared. These resources include computational resources, data storage, networks, catalogs, software modules, and other system resources.

Connectivity layer. The connectivity layer defines the basic communication and authentication protocols required for grid-specific networking-service transactions.

Resource layer. This layer uses the communication and security protocols (defined by the connectivity layer) to control secure negotiation, initiation, monitoring, accounting, and payment for the sharing of functions among individual resources. The resource layer calls the fabric layer functions to access and control local resources. This layer only handles individual resources, ignoring global states and atomic actions across the resource collection pool, which are the responsibility of the collective layer.

Collective layer. While the resource layer manages an individual resource, the collective layer is responsible for all global resource management and interaction with collections of resources. This protocol layer implements a wide variety of sharing behaviors using a small number of resource-layer and connectivity-layer protocols.

Application layer. The application layer enables the use of resources in a grid environment through various collaboration and resource access protocols.

2.1.2 Grid Middleware

Grid middleware provides the services needed to support a common set of applications in a grid environment [6]. It hides the underlying infrastructure details and offers transparent access to the distributed resources, allowing collaborative efforts between organizations.

The middleware sits between the fabric and application layers of the grid architecture, keeping them loosely-coupled with a set of interfaces and protocols. In the bottom of the middleware is the myriad of underlying resources upon which the services are built (local operating systems, networks, file systems, etc.), and the top is where the applications are located.

Standard protocols of grid middleware, which define the content and sequence of message exchanges used to request remote operations, have emerged as an important and essential means of achieving the interoperability upon which grid systems depend. In the late 1990s, grid researchers came together at the Grid Forum, which later became the Open Grid Forum (OGF) [7]. The OGF has been instrumental in the development of the Open Grid Services Architecture (OGSA) [8].

The most commonly used grid middleware are the Globus Toolkit [9] and UNICORE [10]. Latest versions of both embrace the services framework based on the Web Service Resource Framework (WSRF) [11] for the implementation of the OGSA.

The Globus Toolkit

The Globus Toolkit [12] has emerged as a de-facto standard in grid middleware. Initially developed by Ian Foster and Carl Kesselman as a result of I-WAY project [13], Globus now enjoys a large research and development effort. The toolkit has undergone four major revisions, with version 2.4 widely accepted as most stable, and is extensively deployed in the academic community.

Globus is a metacomputing infrastructure toolkit providing basic capabilities and interfaces in areas such as communication, information, resource location, resource scheduling, authentication, and data access. The main components of Globus are: Globus Security Infrastructure (GSI) [14], Globus Resource Allocation Manager (GRAM) [15] and Monitoring & Discovery Service (MDS) [16] [17] .

The Globus Security Infrastructure (GSI) is based on public key concepts (PKI) and X.509 [18] certificates. Each Globus-enabled network host, service, or user has a certificate which is used in authenticating that entity's identity and authorizing access to a resource. All messages communicated between Globus-enabled nodes or components are also secured using Transport Layer Security (TLS) with corresponding certificates. GSI supports delegation of credentials [19] for computations that involve multiple resources and/or sites, thus allowing a single sign-on to use grid resources.

Globus Resource Allocation Manager (GRAM) can be seen as a common interface among all the nodes of a grid. Application requirements, expressed with Resource Specification Language (RSL) [15], are mapped onto local schedulers requests, providing a unique resource identifier contact string which can be used at a later time to query the progress of the job and collect the job's output. GRAM interfaces a wide number of local schedulers, from the simple UNIX fork to Local Resource Management Systems (LRMS) such as Portable Batch System (PBS) [20], Load Sharing Facility (LSF) [21], Condor [22], and Sun Grid Engine (SGE) [23], through a modular architecture. Once the job is submitted, GRAM captures its standard and error outputs, and provides monitoring facilities for proper/improper termination.

The dynamic nature of grid environments forces toolkit components, programming tools, and applications to adapt their behavior in response to changes in system structure and state. Monitoring & Discovery Service (MDS) is designed to support this type of adaptation by providing an information-rich environment in which information about system components is always available. MDS is based on Lightweight Directory Access Protocol (LDAP) [24] components within a hierarchical model. Individual information providers report single metric measurements to a tree of distributed information service servers (called Grid Index Info Server and Grid Resource Info Server). MDS is a central point of contact for locating resources, obtaining their usage statistics and discovering the services that they are able to provide. The information service servers are organized in a hierarchical structure as shown in Figure 2.2.

Each Grid Resource Info Server (GRIS) is registered with one Grid Index Info Server (GIIS), that can in turn be registered with another GIIS. The top GIIS contains information about all the lower levels.

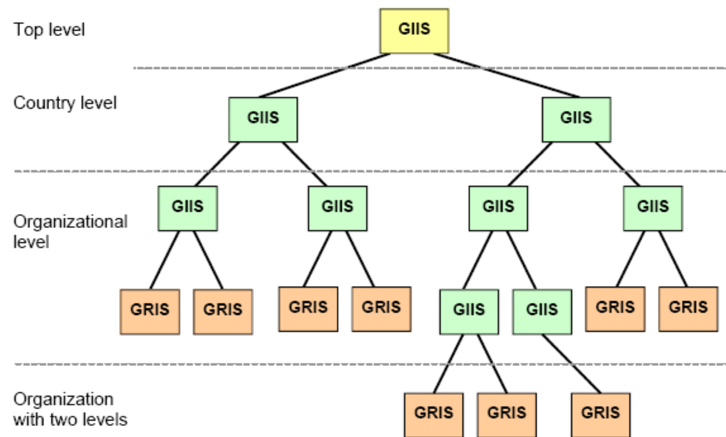


Figure 2.2: Hierarchical MDS structure

Figure 2.3 shows the interaction of the Globus components: the Globus client locates resources using MDS, querying GIIS and GRIS information repositories. With this information, the client contacts the gatekeeper using GSI security services. The gatekeeper implements the GRAM interface and interprets the RSL in the job manager that will finally submit the job to the local resources.

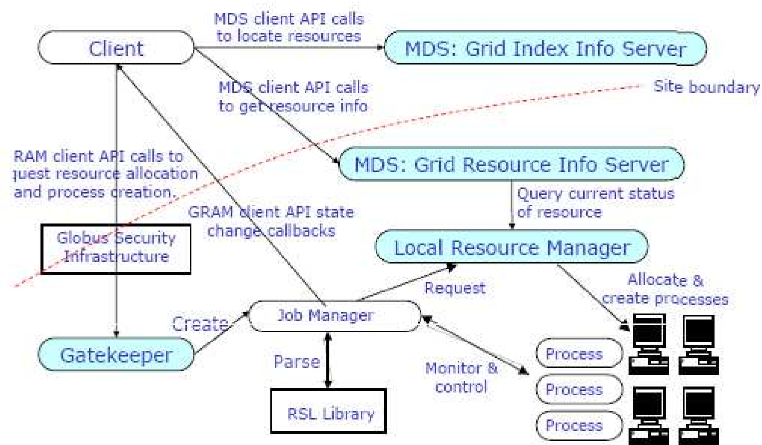


Figure 2.3: Globus components interaction.

Unicore

UNICORE (UNiform Interface to COmputing REsources) is a grid computing technology that provides seamless, secure, and intuitive access to distributed grid resources such as supercomputers or cluster systems and to the information stored in databases. UNICORE was primarily developed by two projects funded by the German ministry for education and research. In various European-funded projects UNICORE has evolved to a full-grown and well-tested grid middleware system over the years. Currently, it is used in daily production at several supercomputer centers world-wide.

Figure 2.4 illustrates the architecture of the UNICORE 6 grid middleware and its Web Services-based interfaces that conform to OGSA concepts. Authenticated end-user requests from different Web Services-based clients pass the UNICORE Gateway and initiate the operations of services deployed within the UNICORE Service Container.

The UNICORE Atomic Services (UAS) are the main interfaces that allow for the exploitation of the core functionality by Web Services-based clients. This core functionality includes job submission and control, file and data transfer, as well as storage management. The UAS consists of several web services. First and foremost, the Target System Service (TSS) models a physical computational grid resource such as a supercomputer. The TSS exposes various pieces of information, for example details about the total numbers of CPUs, memory, etc. and preinstalled applications on the grid enabled resource. Through the TSS, grid jobs described in the Job Submission Description Language (JSDL) are submitted to one UNICORE site. The jobs are controlled by the Job Management Service (JMS). To support data staging for JSDL jobs, the Storage Management Service (SMS) is used to access storage within grid infrastructures.

The function of the Network Job Supervisor (XNJS) as the execution back end is to control and manage the state and persistency of jobs. Hence, one of the major tasks of the XNJS is to parse JSDL documents and form the rather abstract job descriptions into site specific commands. Authorization is accomplished by using the enhanced UNICORE User Data Base (UUDB) in conjunction with extensible policy validations. Then, all commands are forwarded to the UNICORE Target System Interface (TSI) which is directly connected to the already existing batch sub-system (e.g. LoadLeveler, Torque, or LSF) running on the supercomputer.

Many European and international research projects base their grid software implementations on UNICORE. Examples are EUROGRID [25], VIOLA [26], and the Japanese NaReGI project [27]. These projects extended or are extending the set of core UNICORE functions, including new features specific to their research or project focus. The goals of such projects are not limited to the computer science community.

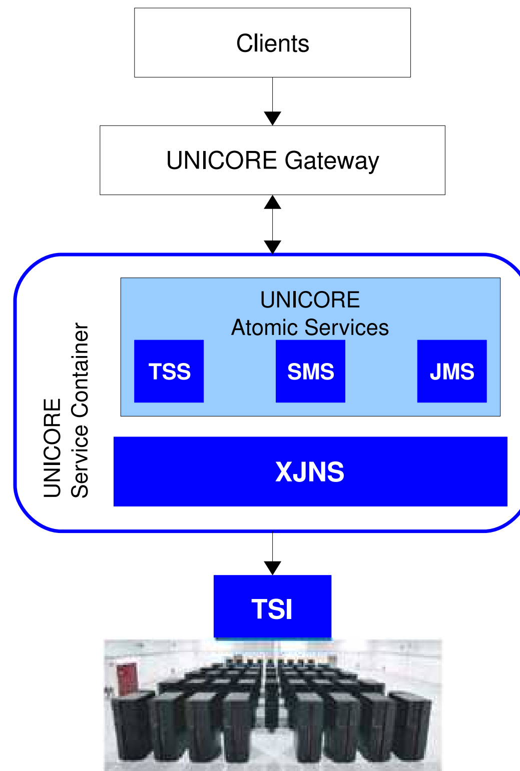


Figure 2.4: UNICORE architecture.

Though both UNICORE 6 and the latest Globus Toolkit v4.2 use the same basic technologies, they have very different security models, basic services and interfaces, and are thus not directly interoperable.

2.2 Grid Infrastructure Initiatives

There are several grid projects going on in the world. Among them, some concentrate on the development of software tools and middleware (as Globus and UNICORE reviewed above), while others are focused on the development of new infrastructure so that scientists can make use of these facilities for their research. Many of these initiatives are motivated by large-scale scientific projects that will involve the production and analysis of data at an unprecedented scale. In this section we review some of the most relevant of such projects.

2.2.1 EGEE

Enabling Grids for E-sciencE (EGEE) is the largest multi-disciplinary grid infrastructure in the world, bringing together more than 120 organizations to provide scientific computing resources to the European and global research community. EGEE comprises 250 sites in 48 countries with more than 68,000 CPUs available to some 8,000 users, 24 hours a day, 7 days a week.

Originally, EGEE used middleware based on work from its predecessor, the European DataGrid (EDG) project [28], and later developed into the LCG middleware stack, which was used on the EGEE infrastructure early in the project. In parallel, EGEE has developed and re-engineered most of this middleware stack into a new middleware solution, gLite, now being deployed. The gLite stack combines low level core middleware, with a range of higher level services. gLite integrates components from current middleware projects, such as Condor and the Globus Toolkit, as well as components developed for the LCG project.

The gLite grid services are organized as "node-types" that ensure easy installation and configuration on the chosen platforms (currently only Scientific Linux versions 3 and 4.) Figure 2.5 shows the shows the basic building blocks of gLite. Each site in the gLite testbed includes the following computational and data storage resources:

- A Computing Element (CE) [29] machine providing the interface between the grid and the local processing farm. It acts as a generic interface to the cluster: a Local Resource Management System (LRMS) (sometimes called batch system), and the cluster itself, a collection of Worker Nodes (WNs), the nodes where the jobs are run.
- An User Interface (UI) machine enabling users to access the testbed. This machine provides Command Line Interface (CLI) tools to perform some basic grid operations.
- A Storage Element (SE) [30] provides uniform access to data storage resources. The Storage Element may control simple disk servers, large disk arrays, or tape-based mass storage systems. Storage Elements can support different data access protocols and interfaces.
- A Monitoring Box (MonBox) collects local data for monitoring and accounting purposes.

Additionally, a set of core node-types provide global collective services:

- Workload Management System (WMS): accepts user jobs, assigns them to the most appropriate resource, records their status, and retrieves their output.

- Information Index (II): the root entry point for an MDS [17] information tree that contains the resource information published by the CE and SE systems. This information is essential for the operation of the whole grid, as resources are discovered due to the II. The published information is also used for monitoring and accounting purposes. Much of the data collected by the II conforms to the GLUE Schema [31], which defines a common conceptual data model to be used for grid resource monitoring and discovery.

- Virtual Organization Membership Service (VOMS) Server: a repository of authorization information used by the testbed systems to manage information about the roles and privileges of users within Virtual Organizations.

- LCG File Catalogue (LFC) [32]: a service that stores information about the location of physical files in the grid. The WMS uses the LFC in order to make scheduling decisions based on the location of the files required by the jobs. It can be used also directly by the user through the UI and the CLI tools.

- RGMA Server [33]: currently used for accounting and both system- and user-level monitoring. It also holds the same GLUE schema information as the II, although it is not currently used to locate resources for job submission.

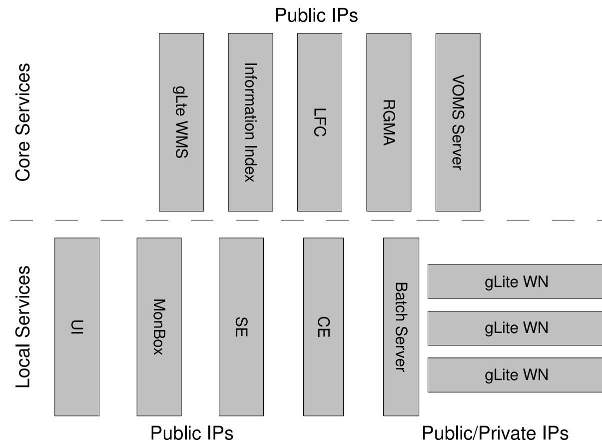


Figure 2.5: The gLite core and local services.

The basic objective of the EGEE project, and also of the preceding EDG project, is to support the distributed processing of very large data volumes, like the experimental data that will be recorded at the Large Hadron Collider at CERN [34]. This processing, and the execution of corresponding simulations, are executed in batch mode. Therefore the gLite middleware stack is oriented to this kind of batch execution of jobs.

2.2.2 OSG

The Open Science Grid (OSG) [35] consortium provides an open grid infrastructure for science in the US. OSG combines resources at many US labs and universities and provides access to shared resources for the benefit of scientific applications.

The OSG architecture is Virtual Organization (VO) based. Most services are instantiated within the context of a VO. The OSG baseline services and reference implementation can support operations within and shared across multiple VOs.

Within OSG, a Site is a set of processing resources, storage resources and services co-located and centrally administered. Sites provide interfaces allowing remotely submitted jobs to be accepted, queued, and executed locally. The priority and policies of execution are controlled both by the VO and the site itself. VO policies are defined by "roles" given to the user through the VOMS service. Site policies and priorities are defined by mapping the user and the user's roles to specific accounts used to submit the job to the batch queue. OSG supports the Condor-G job submission client [36], which interfaces to either the pre-web service or web services GRAM Globus interface at the execution site. Job managers at the backend of the GRAM gatekeeper support job execution by local Condor, LSF, PBS, or SGE batch systems.

Storage Elements are physical sites where data is stored and accessed. Examples are physical file systems, disk caches, and hierarchical mass storage systems. Storage Elements manage storage and enforce authorization policies over who is allowed to create, delete, and access physical files. They enforce local as well as VO policies for the use of storage resources. They guarantee that physical names for data objects are valid and unique on the storage device(s), and they provide data access.

The OSG capabilities and schedule for development are driven by US participants in experiments at the Large Hadron Collider, currently being built at CERN. As in the case of EGEE, it is oriented towards the batch processing of very large amounts of data.

2.2.3 DEISA

The Distributed European Infrastructure for Supercomputing Applications (DEISA) [37] research infrastructure is constituted of a number of leading national supercomputers in Europe, interconnected with a high bandwidth point to point network provided by GEANT [38] and the National Research Networks (NRENs). High bandwidth network connectivity is required to guarantee the high performance of the distributed services, and to avoid performance bottlenecks.

The DEISA consortium follows a hierarchical strategic approach for the deployment

and the evolution of the infrastructure. It is structured as a layer on top of the national supercomputing services, and it coexists with them. This infrastructure addresses the computational challenges that require the coordinated action of the different national supercomputing environments and services for both efficiency and performance, taking into account a few very basic strategic requirements:

- the necessity of the fast deployment of a persistent, production quality supercomputing infrastructure with continental scope
- the coexistence of the European infrastructure with national services, which requires reliability and non-disruptive behavior
- user transparency (users should not be aware of complex grid technologies) and applications transparency (minimal intrusion on applications, which, being part of the corporate wealth of research organizations, should not be strongly tied to an IT infrastructure)

The DEISA Grid incorporates several different processors and operating systems (IBM Linux on PowerPC, IBM AIX on Power4-5, SGI Linux on Itanium, and NEC vector systems). DEISA has deployed middleware based on UNICORE that enables the transparent access to distributed resources, high performance data sharing at a continental scale, and transparent job migration across similar platforms. The next planned actions are the deployment of a co-scheduling service (synchronizing remote supercomputers) and high performance data transfer services across sites.

2.2.4 CrossGrid and Interactive European Grid

The projects described above do not address a specific important point in the e-Science needs of researchers: to generate the final results of many studies, they need to perform complex interactive analysis and simulations that may include visualization, parameters tuning, etc. Both CrossGrid [39] and Interactive European Grid (int.eu.grid) [40] projects aim at enabling such interactive analysis in grid environments.

CrossGrid

The primary objective of the CrossGrid Project (2002-2005) was twofold: to further extend the grid environment to interactive applications, and to extend the effort to 11 European countries. The applications were characterised by the interaction of a person with a processing loop. They required a response from the computer system to an action by the person under a variety of different time scales; from real

time through intermediate to long term, they were simultaneously compute- as well as data-intensive. Examples of these applications are: interactive simulation and visualization for surgical procedures, flooding crisis team decision support systems, distributed data analysis in high-energy physics, and air pollution combined with weather forecasting.

The CrossGrid project addressed user-friendly grid environments. Portal access to the grid infrastructure and user applications, independent of the user location is very important to be practical. As a major result in this area, the Migrating Desktop [41] was developed. The Migrating Desktop is an advanced, user-friendly environment that serves as a uniform grid working environment independent of specific grid infrastructure. It uses a Java-based GUI designed specifically for mobile users, and it is platform independent. It is a complex environment that integrates many tools and allows work with many grids both transparently and simultaneously.

The CrossBroker, was developed as a significant extension to the resource broker originally developed by the European DataGrid Project [42]. The CrossBroker was used in production services of the CrossGrid after being validated by several project partners. This initial release included support for the execution of parallel applications using MPI [43], running either inside a cluster (using MPICHP4 [44])or across different sites (using MPICH-G2 [45]). It also included interactivity services that allowed the execution of remote applications as if they were local.

The CrossGrid international distributed testbed [46] [47] shared resources across sixteen European sites, which ranged from relatively small computing facilities in universities to large computing centers, offering an ideal mixture to test the possibilities of the grid framework. National research networks and the high-performance European network, Geant [38], assured the inter-connectivity between all sites. Figure 2.6 shows a map with the different testbed sites. The CrossGrid testbed included a total of approximately 200 CPUs and a distributed storage capacity above 4 Terabytes.

Interactive European Grid

The aim of the Interactive European Grid (i2g) project [48] is to deploy a production quality e-infrastructure, interoperable with existing environments using gLite, while providing advanced support for scientific applications. Of particular interest are interactivity, parallel execution, and graphical visualization. The project has exploited and consolidated the main CrossGrid achievements, such as the CrossBroker and the Migrating Desktop, to provide intra-cluster support and intercluster support for parallel applications, visualization and handling of video streams to support graphical applications, coupled with a mechanism to support on the fly response to user interaction.

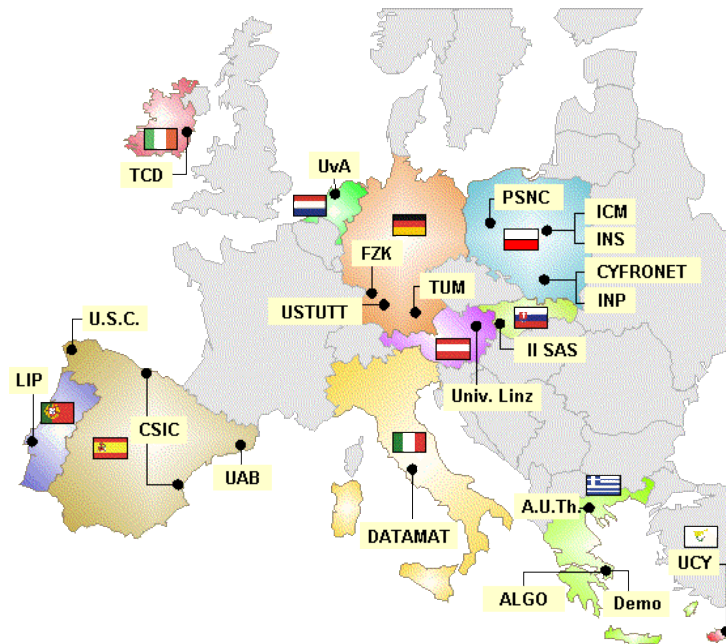


Figure 2.6: The CrossGrid Testbed sites

As in the CrossGrid project, a set of applications that benefit from the deployment of an infrastructure such as that of i2g was identified. This set of representative applications belong to five research areas: high energy physics, nuclear fusion, astrophysics, environmental science, and medical applications. All of them require graphical display and visualization to some extent, and all make heavy use of MPI. Apart from the existing middleware developed previously, new software has been created in order provide the required services for applications.

Two significant new developments in i2g are *mpi-start* [49] and *i2glogin* [50]. *Mpi-start* is a software layer that hides all the heterogeneity inherent to grid hardware and software set up (file system, MPI implementation, etc.), and it allows for flexible and transparent use of MPI parallel applications. The consortium has chosen to adopt OpenMPI [51] as its parallel library implementation, because its modular approach makes it well suited for heterogeneous environments like the grid. However, *mpi-start* can also handle MPICH, as is the case with EGEE, which has quickly adopted this solution for application clusters with a demand for parallel support.

Support for interactivity and visualization is achieved using the *i2glogin* middleware together with the Grid Video (GVid) codec [52]. The *i2glogin* middleware allows fully interactive connections to the grid, with functionality comparable to that of ssh, but without the overhead of a server side running on top of every grid node. The Grid Video, or Gvid, codec encodes the visualization within a video stream,

saving bandwidth, and sends the output to any grid resource, for example to EGEE UIs. It supports mouse, keyboard and GUI events, which are transported over the network by *i2glogin*.

The i2g infrastructure [53] includes two independent testbeds: production and development. The production testbed is composed of clusters at nine different sites (see Figure 2.7), and it intends to provide users with the quality of service they demand. Each cluster deploys different CPU architectures (Xeons, Opterons, Pentium Ds) and uses their preferred Local Resource Management System for job management control. This distributed production infrastructure represents more than 700 CPU cores and 16 TB of storage.

Figure 2.7: The i2g infrastructure map.

The development testbed aims at providing a realistic, yet flexible environment. It includes four clusters and is used to support the testing and integration of the middleware, as well as to allow early access to the new middleware functionality expected to become available in production. The int.eu.grid infrastructure architecture relies on the same basic building blocks as EGEE. The i2g collaboration has enhanced preexisting EGEE services and deployed some additional ones.

2.3 Grid Scheduling

As grids become almost commonplace today, with many projects using them for production runs, grid scheduler (or broker) services are needed to free the users from the cumbersome work of job handling. In traditional computing systems, resource management is a well-studied problem. Resource managers such as batch schedulers and workflow engines exist for many computing environments. These resource management systems are designed to operate under the assumption that they have complete control over a resource and can thus implement the mechanisms and policies needed for effective use of that resource in isolation. Unfortunately, this assumption does not apply to the grid. A grid scheduler must make selection decisions involving resources over multiple administrative domains, in an environment where it has no control over the local resources, where the resources are distributed, and where information about the systems is often limited or dated.

Grid scheduling is defined as the process of making scheduling decisions involving resources contained within multiple administrative domains. This process can include searching multiple administrative domains in order to use a single machine, or scheduling a single job to use multiple resources at either a single site or multiple sites. From a grid point of view, a job is anything that needs a resource. Scheduling, in such an environment, requires a different approach than is presently used in distributed systems. This approach is complex due to various factors, specifically:

Multiple layers of schedulers. Grid resource management involves many players and possibly several different layers of schedulers. At the highest level are grid-level schedulers that may have a more general view of the resources but are "far away" from the resources where the application will eventually run. At the lowest level is a local resource management system that manages a specific resource or a set of resources. Other layers may be between these, for example one to handle a set of resources specific to a project. At every level additional people and software must be considered.

Site autonomy. Computing resources are geographically distributed under different ownerships, each having its own access policy, and cost. Every resource owner will have a unique way of managing and scheduling resources, and the grid schedulers must ensure that they do not conflict with resource owner's policies. The grid scheduler does not have ownership or control over the resources, making the scheduling process harder. Site autonomy and the possibility of failure during allocation introduces a need for specialized mechanisms for allocating resources, initiating computation on those resources, and monitoring and managing those computations.

Shared resources and variance. The resources are not dedicated to the grid scheduler. In most cases, the resources are shared among many users and projects.

Such sharing results in a high degree of variance and unpredictability in the capacity of the resources available for use. The heterogeneous nature of the resources involved also plays a role in varied capacity.

Conflicting performance goals. Grid resources are used to improve the performance of an application. Often, however, resource owners and users have different performance goals: from optimizing the performance of a single application for a specified cost goal, to getting the best system throughput or minimizing response time. In addition, most resources have local policies that must be taken into account. Indeed, the policy issue has gained increasing attention. How much of the scheduling process should be done by the system, and how much by the user? What are the rules for each?

Most systems described in the literature follow a similar pattern of execution when scheduling a job on a grid. There are typically three main phases as described in [54]:

1. Resource discovery. In this first stage of scheduling, a list of potential resources is generated. This phase filters the resources, by taking into account the if the user is authorized to use them and basic job requirements (such as operating system, architecture, RAM).
2. Information gathering and selection. The dynamic information about the discovered resources is gathered in order to select which resource or resources will be used for the execution of the application.
3. Job execution. The last stage includes submission of the job to the selected resource or resources, file staging, monitoring the execution of the application, and cleanup once it has finished.

2.3.1 Grid scheduling systems

Most of the existing grid scheduling systems are focused on the execution of sequential jobs. Here we summarize some of the ones that have appeared in recent years. Although some of them allow the execution of parallel or interactive jobs, none of them consider those jobs as a single entity.

Globus

The Globus Toolkit [12] does not provide the functionality of a Grid Resource Management System, however it is used in many cases for the “manual” scheduling of

jobs. The user performs the scheduler functions: the discovery and selection of sites, submission of jobs to the remote resources, monitoring the execution, and the gathering of the output files.

Globus also includes a mechanism for co-allocation called Dynamically Updated Resource Online Co-allocator (DUROC) [55]. DUROC provides basic co-allocation methods implemented as a set of libraries to be linked with applications and submission tools, and it does not provide resource brokering or fault tolerance. DUROC is used as a building block for grid-enabled MPI implementations such as MPICH-G2 [45].

Condor-G

Condor [22] is a high throughput computing environment that allows users to take advantage of both dedicated and non-dedicated computers. Unlike many other scheduling systems, the use of non-dedicated computers adds complexity to Condor: jobs may be preempted before they have fully completed and the inevitable heterogeneity of the resources available.

Condor-G [36] provides a front-end to a computational grid. It can manage jobs destined to run at distributed sites from a local Condor queue. It provides job monitoring, logging, notification, policy enforcement, fault tolerance, and credential management. Condor-G allows users to specify a single grid site as a destination for jobs. However, when users have a variety of sites to choose from and there is no other resource broker to make the decision, Condor-G can use matchmaking to decide which grid site a job should run on, if the grid sites are advertised in Condor via an external process. Currently, Condor-G is able to interact with different grid middleware, including Globus and UNICORE.

NIMROD/G and Gridbus

Nimrod/G [56] is resource broker for managing and steering task farming applications such as parameter studies on computational grids. It uses an economy-driven model for resource management, and it is used in a framework called Grid Architecture for Computational Economy (GRACE) [57]. Nimrod/G has a hierarchical machine organization and uses a computational market model for resource management. It uses the services of other systems such as Globus and Legion [58] for resource discovery and dissemination. State estimation is performed through heuristics using historical pricing information. The scheduler tries to assign the “cheapest” resource of those available to run each job, by taking into account the job QoS requirements, like deadline and budget.

GridBus [59] is a resource broker developed from Nimrod/G, specializing in jobs that handle large amounts of data. It offers the services of locating data sources as well as access to those sources while the jobs are running.

Nimrod/G and GridBus automatically process the execution of jobs within a grid environment. However, they are focused on parameter sweep applications and data-intensive applications.

AppLeS

The AppLeS [60] project primarily focuses on developing scheduling agents for individual applications on production computational grids. AppLeS agents use application and system information to select a viable set of resources. Applications have embedded AppLeS agents that accomplish resource scheduling on the grid.

The AppLeS framework contains templates that can be applied to applications which are structurally similar and have the same computational model. The templates allow the reuse of the application-specific schedulers within the AppLeS agents. Templates have been developed for parametric and master/slave applications, where each task is an independent, sequential job. No parallel job execution is supported.

The AppLeS scheduler maps jobs to resources, and the problem of allocating those resources is given to a meta-scheduler.

GrADS

The Grid Application Development Software Project (GrADS) [61] provides both programming tools and an execution environment to ease program development for the grid. It replaces the the discrete, user controlled stages of preparing and executing a grid application with an end-to-end software-controlled process. The project seeks to provide tools that enable the user to focus only on high-level application design, without sacrificing application performance.

The GrADS architecture incorporates user problem solving environments, Grid compilers, schedulers, performance contracts, performance monitors, and reschedulers into a seamless tool for application development. As in the case of the AppLeS system, scheduling occurs at the application level.

GridWay

GridWay [62][63] is a meta-scheduler which allows the execution of jobs in diverse grid environments. It performs the resource discovery and selection, job submission, job monitoring and termination, and it has a modular architecture that enables the usage of different grid architectures with a single access point.

gLite WMS

The gLite WMS (Workload Management System) [64] is the evolution of the LCG-RB [65] that was developed within the European DataGrid project [28]. It does grid resource management, specializing in the execution of large batch applications. It takes into account the data requirements of applications. Jobs are considered on a FCFS basis, and the possibility of managing DAGs with Condor DAGMan exists. The gLite WMS has a monolithic architecture that does not allow the execution of parallel jobs. Condor-G is used as a front-end to the remote sites.

KOALA

KOALA [66][67] is a co-allocating scheduler that permits the execution of parallel jobs within a multicluster architecture. Developed to manage the resources of the DAS-2 [68] testbed in the Netherlands, it performs a system level scheduling given its knowledge of the entire system. KOALA includes two policies for the co-allocation of parallel applications: a Close-to-files policy where the location of input files is taken into account, and a Worst Fit policy. Both policies only consider free resources.

KOALA's mechanisms and policies are mostly oriented towards a homogeneous environment such as the DAS-2.

2.4 Execution of Parallel and Interactive jobs on Grids

In the previous section we outlined some of the recent grid resource management systems. Most of them do not consider the execution of parallel and interactive jobs, although users would greatly benefit from this capability. In this section we present the current efforts for executing such jobs on grids, and we review the main challenges.

2.4.1 Parallel Jobs

Grid environments open the possibility of running parallel applications which can efficiently use many processors, taking advantage of the different resources composing a grid. In order to achieve this, a Grid Resource Management System (GRMS) must be able to co-allocate the resources of different administrative domains.

Using resources from different sites creates a highly heterogeneous environment. Both the computing and network resources are heterogeneous. The wide-area communication has a clear impact on the efficiency of the parallel applications.

On the one hand this impact has been studied in recent research of applications designed without considering the use of low latency links between processes. Ernemmann et al. [69] used simulation to compare the performance of co-allocated parallel applications to applications executed within the same cluster. They conclude that the usage of multiple sites can improve the results ,as long as the increase in execution time due to communication overhead is limited to a 25%. The simulations in [70] conclude that co-allocation is beneficial, as long as the number and sizes of job components, as well as the slowdown of application due to the wide-area links are limited.

On the other hand, the development of applications designed to consider wide-area links obtain good results when executing in such environments. In [71], general communication reduction and latency hiding techniques are explored for applications originally intended for a single cluster. The use of hierarchical approaches with master/worker has also been demonstrated to scale and efficiently uses the resources [72] when executed on several clusters.

Message Passing Interface (MPI) [73] is a widely used standard library for parallel application communication. Many implementations of MPI exist, and most of them are designed for clusters and supercomputers. Open MPI [51] and MPICH [44] are widely adopted by the high performance computing community and are two of the most important today.

Open MPI is a recent, open source MPI-2 implementation centered around component concepts. It is developed by a consortium of research, academic, and industry partners using prior research from the LAM/MPI [74], LA-MPI [75] and FT-MPI [76] projects. Its component architecture enables the run-time composition of independent software add-ons, supporting a heterogeneity of networks, job schedulers, and operating systems.

MPICH is a freely available, portable implementation of MPI developed at Argonne National Laboratory. It is a complete implementation of the MPI-1 standard, and it includes most of MPI-2. Its main goal is to provide an MPI implementation that efficiently supports different computation and communication platforms while

maintaining portability.

The heterogeneity of grid environments is also considered by other MPI implementations, such as MPICH-G2 [45], PACX-MPI [77], GridMPI [78], and MagPIe [79].

MPICH-G2 is the most prominent of the grid computing implementations. It is a complete implementation of the MPI standard using the MPICH implementation as its basis. MPICH-G2 hides heterogeneity by using the Globus Toolkit services for such purposes as authentication, authorization, executable staging, process creation, process monitoring, process control, communication, and remote file access. The CrossBroker manages the submission of MPICH-G2 applications by handling the executable staging, process creation, monitoring, and control of applications. The use of MPICH-G2 is limited to sites where the Globus middleware is installed. The major limitation of MPICH-G2 is the need for public IP addresses on the machines involved in the computation.

PACX-MPI was developed in order to allow one MPI application to be run on varying hardware architectures. The software adds two additional MPI processes for each cluster that do not execute the user's application code: process 0, in charge of incoming communication, and process 1 in charge of the outgoing communication. The communication pattern of a broadcast in PACX-MPI is shown in Figure 2.8. In the Figure, two sites, A and B, are executing a PACX-MPI application with 8 user processes that are numbered globally (grey box numbers). Two additional processes are added at each site to handle communication. The white boxes show the local MPI process number for each of the sites. When the processor number 3 (global rank) performs a broadcast, it sends data to the local processes, including the local process number 1. This process in turn sends the data to the remote site. Data arrives at local process 0 in site B and will be distributed within that site. The communication between the clusters is done over TCP connections.

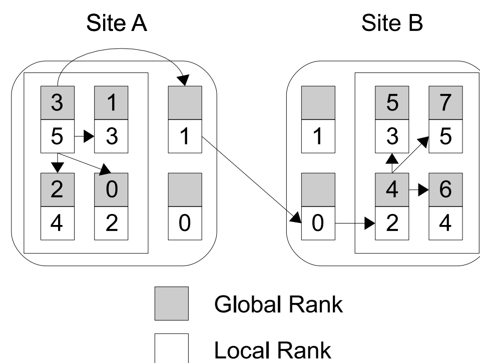


Figure 2.8: Pacx Communication Structure

PACX-MPI does not have external dependencies and simply requires recompiling the MPI code with the corresponding wrappers, thus allowing the use of highly efficient MPI implementations in each of the clusters and handling the interconnectivity with only one entry point. These characteristics make PACX-MPI well suited for grid sites where the machines use private IP addresses. PACX-MPI includes a mediator process, called startup-server, that handles the initial connection required among the various clusters that comprise and execute the application.

2.4.2 Interactivity in Grid Environments

The execution of interactive applications within grid environments has received little attention to date. However, users would like to see the intermediate results of their applications in minutes rather than in hours, even if they have to process a significant data volume or apply complex algorithms. The experience of several projects prove that researchers can benefit from access to powerful resources in an interactive mode at the final stage of their analysis in a wide range of applications [80][81][82][83].

There are currently several mechanisms (such as SSH [84] or VNC [85]) that can be used to create interactive sessions between remote machines. Unfortunately, these mechanisms are not generally suitable for grid environments due to performance and/or administrative limitations. The use of VNC, for instance, introduces a substantial overhead, which results in a slow and sometimes unreliable graphical display when used in a wide area network. The standard distribution of SSH is not grid-enabled, and it does not support grid-related authentication. Even the use of an additional package [86], which adds support for the Globus Grid Security Infrastructure (GSI), does not solve this problem, because users do not have personal accounts on the remote machines where their jobs may run. Typically, computing nodes are managed through a local batch system, but such machines do not have accounts that are available to external users.

The possibility of running a user's application in the near future has been explored in [87], where the creation of a special Globus jobmanager with interactive response is proposed. This approach is limited as all sites must adopt the solution, and in the case of high occupancy of resources it cannot provide appropriate services. The CrossBroker provides interactive response without special requirements on the remote sites.

The execution of interactive applications in grid environments is treated in [88], where the use of virtual machines is proposed in order to run applications. The problem of allocation of the virtual machines is not considered.

In [89], an architecture for the execution of interactive applications is presented. The architecture is focused on the creation of interactive sessions at the remote resources,

and the resource scheduling is not considered.

2.5 Contributions

The execution of parallel and interactive applications in a grid environment arises from a set of challenges that have not been addressed completely before. Most of the Grid Resource Management Systems are devoted to the execution of serial batch applications or single-site parallel applications. The scheduling of interactive jobs is not addressed by current systems. In this work we propose a comprehensive solution that deals with both interactive and parallel jobs. The main contributions of this work are the following:

1. The definition of an architecture for the specification and execution of parallel and interactive jobs. This architecture deals with the heterogeneity of resources, applications, parallel library implementations, and an interactive mechanism by using different modules: Application Launchers, Job Starters and Interactive Agents. The combination of these modules allows the execution of a wide range of applications within grid environments.
2. A job definition language that allows the specification of jobs conforming to the proposed architecture. The language is an extension of the semi-structured JDL language from the European Data Grid project.
3. A mechanism for multiprogramming in grid environments. This mechanism allows the fast start of interactive jobs, even under conditions of high occupancy on the resources, and the co-allocation of parallel jobs without the need for reservation mechanisms on the resources.
4. An implementation of the architecture and the multiprogramming mechanism in the CrossBroker GRMS. The implementation leverages existing efforts in the area by taking advantage of already available components. The CrossBroker has been used as grid scheduler in European grid initiatives as already presented.
5. An experimental study where we measure the benefits of the proposed mechanisms in grid environments. We also measure the overhead introduced by the different components of the system.

2.6 Conclusions

In this chapter we have introduced the required background material to read the remainder of this work. Grid computing is an important platform for the next gen-

eration of scientific applications, where new challenges arise due to the distributed and heterogeneous nature of the environment. The middleware is an essential part of the architecture on this computing platform, with Globus being the current de-facto standard middleware. We have also presented the two grid initiatives in which this work was developed. Grid Scheduling Systems free the user from the task of job handling. Traditional resource management systems operate under the assumption that they have control over the resources, however this does not apply to grid environments. We have outlined some of the recent work in grid scheduling and we have reviewed the different approaches for running parallel and interactive jobs in grids.

CHAPTER 3

An Architecture for Parallel and Interactive Jobs

In this chapter we propose an architecture for managing interactive and parallel jobs in a grid environment. In the first section, the relevant components for the execution of jobs within the grid environment are described. In Section 3.2, we present a model for both the jobs and the components for their execution within grids. A language that allows the specification of such jobs is presented in Section 3.3. Finally, in Section 3.4, the architecture for a Grid Resource Management System (GRMS) that schedules these parallel and interactive jobs is described.

3.1 The Grid Environment

Our system model is related to several projects that share similar basic middleware such as EGEE, OSG, or int.eu.grid. The system (or testbed) for those projects was outlined in Section 2.2. Here we describe the components relevant to job management in such environment. It should be noted that the system architecture is generic enough to be extended to other kinds of testbeds based on different middleware. Figure 3.1 diagrams the architecture of the environment.

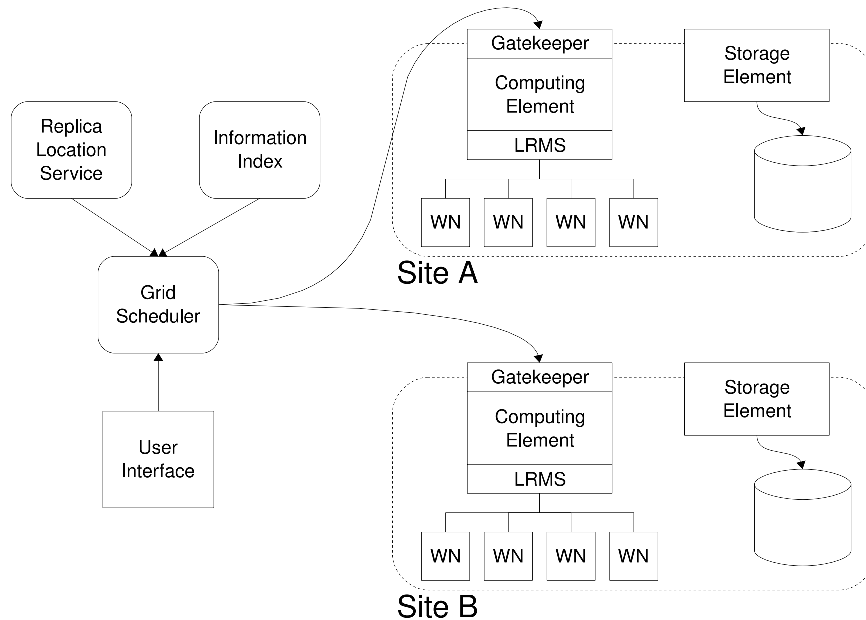


Figure 3.1: Grid environment architecture

Every site (or resource[1]) has an entry point or *Computing Element* (CE) where a Globus Gatekeeper [15] is installed. The Gatekeeper sits between the local compute farm and the external users. Jobs are submitted from a Grid Scheduler to the Globus Resource Allocation Manager (GRAM) service. The GRAM service in turn submits the jobs to a Local Resource Management System (LRMS) such as PBS, Condor, or SGE. The LRMS manages a set of *Worker Nodes* (WN) where the jobs are actually executed. The gatekeeper also has a Globus Monitoring & Discovery Service (MDS) information system that publishes information about local resources and user authorization to a central information repository called the *Information Index* (II). The information published in the Information Index follows the Glue Schema [31] specification. The Glue Schema is proposed by the Open Grid Forum (OGF) [7] as the standard representation model for resources in grid environments. It includes both static attributes (such as number of CPUs, system architecture, LRMS, and available software) and dynamic attributes (such as system load, number of jobs running or waiting, number of free CPUs, and estimated response times).

Storage resources at each site are handled by a *Storage Element* (SE). SEs are accessed via standard protocols such as gridFTP [90] or RFIO [91]. Files stored within the SEs are accessed using unique grid identifiers called a GUID (Grid Unique IDentifier) or via human readable aliases called Logical File Names (LFN). Translation

[1]Throughout this work, we use the terms “site” and “resource” interchangeably

from a GUID or LFC to a physical location is performed using a central Replica Location Service (RLS).

The job submissions are made from a *User Interface* that connects to a grid scheduler. This grid scheduler performs all the actions needed in order to execute a job without any further user intervention.

3.2 The Job Model

There are two job classifications in our model: structural and functional. The first classification is determined by the structure of the application such as the number of CPUs it uses and/or how these CPUs are spread across the grid. This structure is not malleable, and it cannot change during the job's runtime. The functional classification is determined by the way the job interacts with the user during its runtime.

Jobs may be classified according their structure under the following types:

Normal. Executed on a single resource and requests a single CPU.

Parallel. Uses more than one CPU at the same time. Parallel jobs consits of one or more job components, each of them executed in a set of CPUs.

Workflow. Consists of a set of inter-dependent jobs, where information or tasks are passed from one job to another for action, according to a set of rules. The jobs that comprise a workflow can be parallel or normal depending on the CPUs needed by each of them.

Parallel jobs can be further classified. The different cases are shown in Figure 3.2 and discussed below:

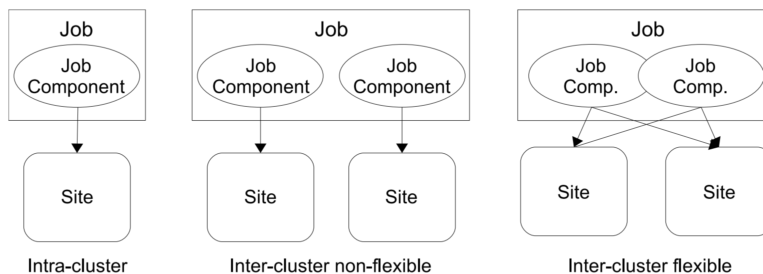


Figure 3.2: Parallel job types

1. Intra-cluster: these parallel jobs have a single component that is executed on a single cluster. All CPUs requested for the job must be allocated within the same resource.

2. Inter-cluster: these jobs consist of one or more components that can be executed over different clusters simultaneously. Depending on the component flexibility, we can differentiate between Flexible or Non-Flexible jobs:

   - Flexible job: the job only specifies the total number of processors it requires. It is left to the scheduler to split up the job if necessary and to decide on the number of components and the number of processors for each component.
   - Non-Flexible job: the job specifies the number of components and the number of CPUs for each component. This job classification is useful for applications that require specific resources.

Jobs have one of two functional classifications:

Batch. jobs that are set up so that they can be run to completion without human interaction, and so that all input data is preselected through scripts or command-line arguments.

Interactive. jobs that require the interaction of the user during execution. This interaction includes the steering of results while the job is running or the modification of the application parameters and on-line behavior.

Both structural and functional classifications are combined in order to define one job completely. Table 3.1 depicts the possible combinations. The only job type that does not allow interactivity is workflow.

	Normal	Parallel	Workflow
Batch	X	X	X
Interactive	X	X	

Table 3.1: Job structural and functional types

In order to support all combinations in our architecture, Job Starters and Application Launchers handle the structural requirements of the jobs, while Interactive Agents manage the functional requirements. They are described in detail in the following sections.

3.2.1 Job Lifecycle

The lifecycle of a job is defined by a finite state machine shown in Figure 3.3. The possible states are:

Submitted The job has been submitted to the scheduler.

Waiting The job has been received by a grid resource scheduler but does not have resources assigned.

Ready The job has a list of execution resources and is submitted to the selected resources.

Scheduled The Computing Element (or set of Computing Elements) has received the job. The job is now in the LRMS queues.

Running The job is running at the remote sites.

Done Job has finished.

Cleared The user has received all of the job's output files.

Aborted The job has been aborted due to an error during its execution or submission phases.

Canceled The user has canceled the job.

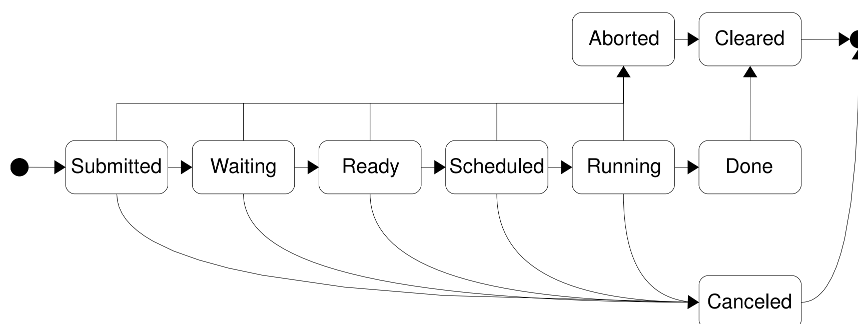


Figure 3.3: Job Lifecycle.

3.2.2 Job Starters and Application Launchers

Different types of applications have different characteristics that must be taken into account when executing them on grid environments. The change rate of grid technology and the differences between systems makes it impossible to have a universal

way to start every job. Therefore, we introduce in our architecture the concept of Job Starters (JS) and Application Launchers (AL), which hide the heterogeneity of the grid and the applications and provides an easy extension mechanism for new applications and systems, with minimal impact on the already existing framework. A Job Starter and Application Launcher pair exists for every type of job supported by the system.

Job Starters are responsible for initiating the applications at the Worker Node level. They handle the details of dealing with the LRMS, as well as the application details such as parallel communication library initialization or application invocation. Application Launchers manage the grid level start up of applications utilizing a global view of the job at the level of the grid scheduler. They assure that all the components of a parallel job are submitted properly to the remote sites, handle any synchronization procedures needed for the job, and monitor the application's execution. Wide-area communication between job components is also set up by the Application Launcher. It supports these application types by knowing how to link the different components among them using the communication libraries provided by the applications.

Figure 3.4 shows the relationship between the Application Launcher, the Job Starters, and the user applications. In the Figure only one site is shown, but in the execution of one job there will be the same number of sites as job components. The Application Launcher submits the Job Starters to the all the Computing Elements of the Sites involved in the execution of the application, as illustrated by arrow (1) in Figure 3.4 for one CE. The CEs will in turn allocate the local nodes, start the Job Starter, shown in the Figure by arrow (2). This Job Starter deals with the specific LRMS and application properties, and initiates one job component as illustrated by arrow (3). This job component contacts the Application Launcher as illustrated by arrow (4) in the Figure, in order to perform the synchronization and set up steps needed to execute the entire application properly.

Application Launchers also deal with failures that may appear during job execution. Both application errors and resource errors are handled. For some application types, failures of components can be tolerated to some degree. Some failed components can even be restarted on different sites without affecting the execution of the job. On the other hand, for some application types such as parallel applications, the failure of a single component can cause other components, or even the whole application, to fail.

Additionally, application level scheduling may be implemented at the level of the Application Launcher, mapping the application components to the resources, while taking into account both resources and application characteristics.

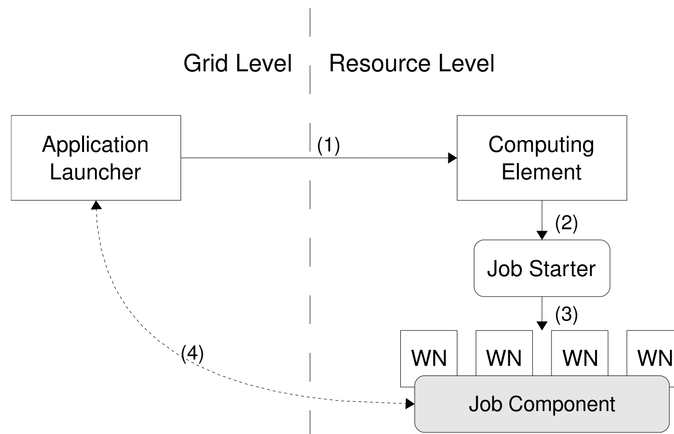


Figure 3.4: Job Starters and Application Launchers

3.2.3 Interactive Agents

Although grid applications are executed on remote sites, the input/output of such applications might be controlled locally, that is, from the submitting machine. This way, users can interact with their applications while they execute remotely. In our architecture we propose a split execution system, where an interactive session is created between two software components: an *agent* and a *shadow*, as shown in Figure 3.5. The agent traps the input and output of an application, and forwards them to a shadow process on another machine via the network. Under this arrangement, a program can run on any networked machine and still execute as if it were running on the same machine as the shadow.

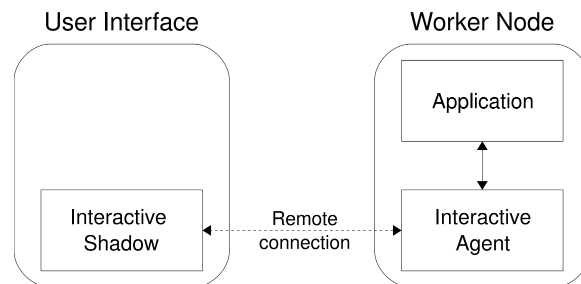


Figure 3.5: Interactive Shadow and Interactive Agent

No restrictions are imposed on the interactive agent or shadow communication schema, but their implementation must take into account the system environment where

they will be executed. At most sites, inbound network connections are forbidden and outbound connections are allowed, but only within a limited range of ports. The Job Starter is in charge of invoking the interactive agent at the moment required by the application, and the interactive agent, in turn, is in charge of starting the application. Figure 3.6 shows the interaction schema between Application Launchers, Job Starters, and Interactive Agents. The user starts the Interactive Shadow in his local machine and submits the job to a grid scheduler, illustrated by arrow (1) in the Figure. Then the scheduler selects the sites for the execution and starts the specific Application Launcher to submit the subjobs to each site. This is illustrated by arrow (2) in Figure 3.6. In this example, two sites will be used, one with processes from 0 to K and the other with processes from K+1 to N. The Job Starter will start the job components in each site and will run the Interactive Agent at the first site (the number of Console Agents depend on the application model and the JobStarter must be aware of such details). Once this synchronization phase has finished, the application can run and the Interactive Shadow and Interactive Agent will initiate communication between them. This is illustrated by arrow (3) in the figure.

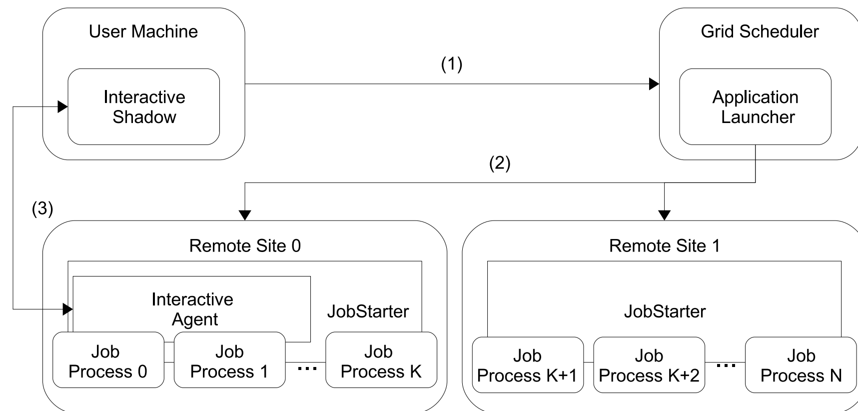


Figure 3.6: Interactive Job Execution

3.3 Job Description Language

Users describe their applications using the Job Description Language (JDL) [92]. The JDL is a Condor ClassAd [93] based language, where a set of attributes define the application to be executed. Originally developed in the European DataGrid project [28], we have extended the format to allow the expression of the specific features of our architecture for interactive and parallel jobs.

A ClassAd is a semi-structured data model, where a finite number of distinct at-

```
Type                  = "Job";
JobType               = "Normal";
Executable            = "my_app";
Arguments             = "-n 356 -p 4";
StdOutput             = "std.out";
StdError              = "std.err";
InputSandBox          = {"my_app"};
OutputSandBox         = {"std.out", "std.err"};
Requirements          = other.GlueHostBenchmarkSI00 >= 1000;
Rank                  = other.GlueHostFreeCPUs;
InputData             = {"test-input.txt"};
ReplicaCatalog        = pcrc.cern.ch;
```

Figure 3.7: JDL job description.

tribute names are mapped to expressions. The lack of a specific schema allows the use of this language in heterogeneous environments such as grids. Moreover, the possibility of arbitrarily nesting descriptions leads to a natural language for expressing resources and job aggregates or co-allocation requests. The ClassAd language folds the query language into the data model, and therefore job requirements may be expressed as attributes of the job.

The JDL is a ClassAd based language with a predefined set of attributes that express the requirements of a job to be executed in a grid environment. Figure 3.7 depicts an example of a simple job using the JDL.

A complete description of the JDL can be found in [92]. Here are description of relevant ones for normal jobs:

- `Type`. This a string representing the type of the request described by the JDL. The possible values are `Job` for normal and parallel jobs and `DAG` for workflows.

- `JobType`. String representing the type of the job described by the JDL. The possible values are `Normal` and `Parallel`, and the default value is `Normal`. In the original JDL specification, this attribute included information about the job implementation or functional characteristics of the job, such as interactivity or checkpointing.

- `Executable`. This is the name of the user executable. It can be an executable pre-installed at the Worker Nodes or transferred from the submitting machine if it is included in the `InputSandbox` of the job.

- `Arguments`. A string that contains all of the job command line arguments.

- `StdInput`, `StdOutput`, and `StdError` are the names and paths of the files that the application will use as standard input, output and error respectively. Wildcards are not allowed. The value specified for `StdError` can be the same as the one for `StdOutput`.

- `InputSandBox`. This is a string or a list of strings identifying the set of files on the user's local disk needed by the job for running, and hence needed to be transferred to the Worker Node. The `InputSandbox` file list cannot contain two or more files having the same name (even if on different paths), as when transferred on the WN they would overwrite each other.

- `OutputSandBox`. This list of strings identifies the list of files generated by the job on the WN, which need to be retrieved and copied back to the user's machine.

- `Requirements`. This attribute represents the job requirements for resources. It is a boolean expression that uses C-like operators. The requirements expression can utilize attributes that describe the resource prefixed with `other.`. All these attributes should conform to the Glue Schema [31] that describe the Computing Elements, since that format is used in the information repositories of the system as stated in 3.1. This expression is evaluated during the scheduling of the job. In the example depicted in Figure 3.7, the job requirement with the `other.GlueHostBenchmarkSI00 >= 1000` expression requires resources which have a SpecInt 2000 benchmark of 1000 or larger.

- `Rank`. This is a floating-point expression that states user preferences on the resources available to run the jobs. A higher numeric value equals a better rank. As with `Requirements`, the `Rank` expression can utilize attributes that describe the resources using the Glue Schema. In Figure 3.7 the job prefers resources with more free CPUs using the `other.GlueHostFreeCPUs` expression.

- `InputData`. A list of files that the job will use during its execution. These are logical file names that are stored in the grid storage systems and can be located using replica location tools. The scheduler should try to minimize the transfer time of these files by allocating sites near the physical location of the files. In the example of Fig. 3.7, the application will use a file called `test-input.txt`.

- `ReplicaCatalog`. This attribute lets the job specify a particular catalog when searching for the file replicas listed in the `InputData` attribute. In the example of Figure 3.7, `pcrc.cern.ch` will be used as catalog.

3.3.1 Extended JDL

The definition of parallel and interactive jobs considered in the `JobType` attribute of the original JDL lacked flexibility and did not allow the specification of our job

model correctly. Therefore we have introduced a set of extensions that enable the model's complete specification of jobs. These extensions can be classified into three types:

1. Specification of parallel jobs,
2. specification of interactive jobs, and,
3. specification of workflows.

Parallel jobs

Parallel jobs need additional attributes. In Figure 3.8 a parallel job description is shown. The `NodeNumber` attribute allows users to specify the number of nodes on which their application will run. The `SubJobType` specifies the type of parallel application used.

```
Type                     = "Job";
JobType                  = "Parallel";
NodeNumber               = 23;
SubJobType               = "pacx-mpi";
Executable               = "my_app";
Arguments                = "-n 356 -p 4";
StdOutput                = "std.out";
StdError                 = "std.err";
InputSandBox             = {"my_app"};
OutputSandBox            = {"std.out", "std.err"};
Requirements             = other.GlueHostBenchmarkSI00 >= 1000;
Rank                     = other.GlueHostFreeCPUs;
```

Figure 3.8: Parallel JDL job description.

The `SubJobType` attribute allows the automatic selection of a Job Starter and Application Launcher for the job. The possible values of this attribute depend on the final implementation of the Grid Resource Management System, and the types of parallel jobs it supports. Currently, the allowed values for this attribute are:

1. openmpi. Defines an application linked with Open MPI;
2. mpich. An application linked with the MPICH library (ch_p4 device);
3. pacx-mpi. An application that can run over multiple sites with PACX-MPI;

4. mpich-g2. An inter-cluster application using the MPICH-G2 library;
5. plain. Intended for advanced users, the `plain` type allows the execution of an application that does not use any of the other MPI implementations. The scheduler will select an Application Launcher suitable for intra-cluster applications, or the user may specify a particular implementation with the `ApplicationLauncher` attribute. Additionally, the user can specify the Job Starter using the two additional attributes: `JobStarter` and `JobStarterArguments`.

Parallel inter-cluster applications are treated as flexible applications by default. If the user wants to specify a non-flexible parallel application, the `Subjobs` attribute in the JDL can be used as illustrated in Figure 3.9. This attribute contains a list of ClassAds, which contains at least the `NodeNumber` attribute specifying the number of nodes for the job component. Optionally, `Requirements` and `Rank` expressions can be included in the component description. The Figure shows an example of an inter-cluster application requiring 10 CPUs, 5 of them in a cluster with machines that offer OpenGL support, and 5 of them in a cluster where machines have more than 1 GByte of RAM, preferring sites with lower estimated response time.

```
SubJobs = {
  [
     NodeNumber = 5;
     Requirements = Member("OpenGL",
                 other.GlueHostApplicationRunTimeEnvironment);
  ]
  [
     NodeNumber = 5;
     Requirements = other.GlueHostMainMemoryRAMSize >= 1024;
     Rank =- other.GlueCEStateEstimatedResponseTime;
  ]
}
```

Figure 3.9: SubJobs specification in JDL.

Interactive jobs

Independent of the type of application (parallel or sequential), users may require interaction with the job during its execution. The input and output of the application is redirected using an Interactive Agent. A boolean `Interactive` attribute in the JDL declares the job as interactive when necessary. The scheduler uses this information in order to use priority scheduling for such jobs. Additionally, the user must specify

the agent with the `InteractiveAgent` and `InteractiveAgentArguments` attributes. The first attribute specifies the type of agent that the job will use, while the second one defines any additional information needed for the execution of the job, such as the location of the user's machine. Figure 3.10 shows an example of a 5 CPU Open MPI interactive job that uses *glogin* [94] as its Interactive Agent. *glogin* is a tool that allows the creation of interactive shells at remote sites using Globus services. It has been adapted for use in our architecture as an interactive agent. The arguments for this agent are the machine and port where the Interactive Shadow is listening for communication with the agent; in this case the IP address is 158.109.65.150 and the port is 24353. The Interactive Agent is included in the `InputSandbox`, but it could also preinstalled be at the remote machine.

```
Type                      = "Job";
JobType                   = "Parallel";
NodeNumber                = 5;
SubJobType                = "openmpi";
Executable                = "interactive_mpi";
Arguments                 = "-n 356 -p 4";
InteractiveAgent          = "glogin";
InteractiveAgentArguments = "-r 158.109.65.150 -p 24353";
InputSandBox              = {"interactive_mpi", "glogin"};
Rank                      = other.GlueHostFreeCPUs;
```

Figure 3.10: Interactive JDL job description.

Workflows

There are many complex applications that consist of inter-dependent jobs that cooperate to solve a particular problem. The completion of a particular job is needed in order to start the execution of jobs that depend on it. This kind of application workflow may be represented in the form of a DAG – a directed acyclic graph. A DAG is a graph with one-way edges that may not contain cycles. It can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies of these programs. Figure 3.11 presents an example DAG that consists of 4 nodes on 3 levels. The execution of the indicated DAG consists of three successive steps:

1. Execution of the node NodeA at the first level.
2. Parallel execution of two nodes (NodeB1, NodeB2) at the second level. The

execution can start if and only if the execution of NodeA (on which these nodes depend) is successful.

3. Execution of the last node (NodeC) from the third level. The execution can start if and only if the execution of all nodes at level two (on which this node depends) is successful.

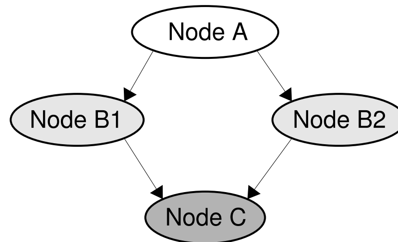


Figure 3.11: Example DAG.

The description of a workflow has two elements: the specification of dependencies between computations (node dependencies) and the specification of each computation (node description). An example JDL file for this workflow is specified in Figure 3.12.

In general, each dependency is specified as a pair of elements positioned between braces, signifying that the second element depends on the first. The first element, as well as the second, may be formed by a set of elements written in brackets. This indicates a dependence of many-to-one, one-to-many or many-to-many elements. Therefore, more than one specification exists for this example DAG:

- {{NodeA,NodeB1}, {NodeA,NodeB2},
  {NodeB1,NodeC}, {NodeB2,NodeC}}
- {{NodeA,NodeB1}, {NodeA,NodeB2}, {{NodeB1,NodeB2},NodeC}}
- {{NodeA, {NodeB1,NodeB2}}, {{NodeB1,NodeB2},NodeC}}

The attribute `Nodes` contains the list of nodes that form the DAG. Each node represents a job to be executed and contains node-specific attributes, as well as the job specification. The attributes for a node's description are:

- `node_retry_count` – specifies how many times a node execution may be retried in the case of failure. This attribute is optional. If this particular node fails, it will be automatically retried as many times as the value specified in this attribute. If undeclared, this attribute will be set to the default value.

```
/* DAG that consists of 4 nodes */
Type = "DAG";
Nodes = [
   Dependencies={ {NodeA,{NodeB1,NodeB2}},
                  {{NodeB1,NodeB2},NodeC} };
   NodeA = [
      description = [
         Executable    = "jobA.sh";
         StdOutput     = "std.out";
         StdError      = "std.err";
         InputSandbox  = {"jobA.sh"};
         OutputSandbox ={"std.out", "std.err"};
      ];
   ];
   NodeB1 = [
      node_retry_count = 3;
      app_exit_code    = { 10, 11 };
      file             = "jobB1.jdl";
   ];
   NodeB2 = [
      file = "jobB2.jdl";
   ];
   NodeC = [
      file = "jobC.jdl";
   ];
];
```

Figure 3.12: JDL Dag description.

- `app_exit_code` – specifies the possible exit codes for a job. If a node fails because of application failure (e.g. segmentation fault, division by 0, a file already registered in a Storage Element), then the entire job should be aborted. However, when a node fails because given resources fail (e.g. a machine failure, LRMS queue problems), this node should be retried automatically. By default, in both cases, the node will be retried `node_retry_count` times. The attribute `app_exit_code` provides the ability to set the job exit code and terminate job execution in case of failure. If the job execution returns one of the specified values, the node will not be retried. Otherwise, the job will be retried automatically in accordance with `node_retry_count`. This attribute is optional.

- `description/file` – A specification of the job. A job can be normal or parallel. There are two ways to specify a job: via an attribute called `description`, where a job is specified directly by this attribute in the JDL, or via an attribute called `file`, where a job is specified in the indicated JDL file. Interactivity is not allowed in workflows.

3.4 CrossBroker Grid Scheduler

The grid scheduler service in our architecture is called CrossBroker. When users submit their applications, our scheduling services are responsible for optimizing scheduling and node allocation decisions on a user basis. Specifically, it carries out three main functions:

1. Select the "best" resources for a submitted application to use. This selection takes into account the application requirements, as well as ranking criteria used to sort the available resources.

2. Perform a reliable submission of the application to the selected resources. This involves the proper co-allocation of resources when the application is distributed among multiple sites.

3. Monitor the application's execution and report on job termination. Once the job is finished, do the necessary clean-up steps at the remote site.

Figure 3.13 presents the main components that constitute our resource management services and the relationship with the system architecture presented in Section 3.1. A user submits a job to a Scheduling Agent (SA) through a User Interface. The job is specified by a ClassAd using the Job Description Language (JDL) already described.

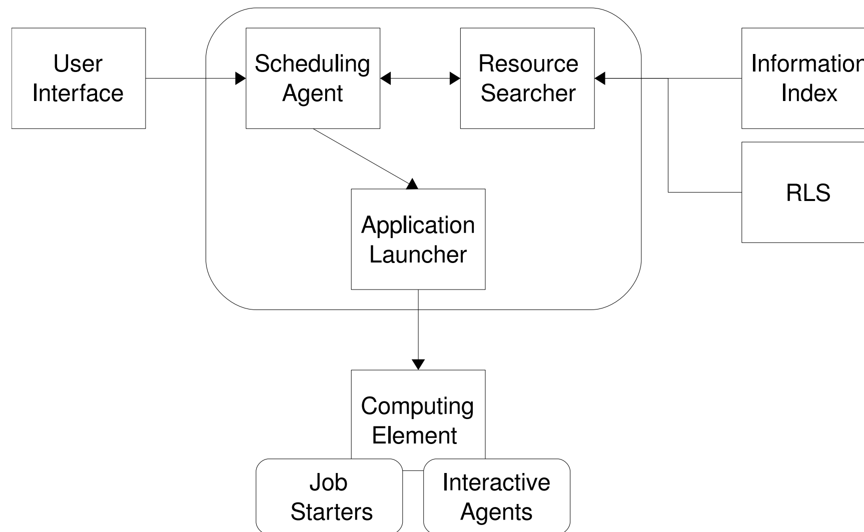


Figure 3.13: CrossBroker Architecture

Once the job has reached the SA, the Resource Searcher (RS) is asked for resources to run the application. The main duty of the RS is to perform the matchmaking between job needs and available resources. Using the job description as input, the RS returns as output a list of possible resources which meet the job requirements. Information about the resources is gathered from the II services, while information about file locations is gathered from the RLS services.

The SA then selects the best resource (or group of resources) from the list returned by the RS. The computing resources (or group of resources) are passed to the Application Launcher, which is responsible for the actual submission of the job. Due to the dynamic nature of the Grid, the job submission may fail on that particular site. Therefore, the Scheduling Agent will try other sites from the returned list until job submission succeeds. The Application Launcher is also in charge of the reliable co-allocation of parallel applications on the Grid. Job Starters handle the execution of the job at the resource level and Interactive Agents provide channels for I/O forwarding from the remote machines to the user and vice versa.

3.4.1 A mechanism for multi-programming

Ideally, interactive applications should always run soon after submission. However, there may be situations where not all of the remote resources involved in an execution are available, causing the ready resources to stay idle until all the subjobs start. We have introduced a time-sharing mechanism that enables both interactive and batch jobs to share a single machine, in such a way that the interactive application starts

its execution as soon as it is submitted (unless all resources in the Grid are busy executing other interactive applications) and proper co-allocation is ensured.

This time-sharing scheme is based on the transparent submission of job agents for jobs submitted by the user. The agent gains control of remote machines independently of the local-site job manager. Each machine acquired by our agent is configured as two virtual slots, in order to create a separate group of dedicated resources for two types of application: batch and interactive. It is worth noting that our concept of slots is lightweight and does not correspond to the classic view of virtual machines that presents the image of multiple operating system configurations (completely isolated from each other) sharing a single machine. In our case, the machine only runs one OS, but we split the machine into two separate execution slots. Each slot contains the executable and files required by the corresponding job. From a logical point of view, batch jobs will run on one virtual slot and interactive jobs will run on the other. However, our agent guarantees that interactive jobs will be executed at a higher priority than batch jobs. When the interactive job is finished, the original priority of the batch job is restored and after completion of the batch job, the agent leaves the machine.

Figure 3.14 illustrates the possible scenarios that CrossBroker deals with for intra-cluster applications:

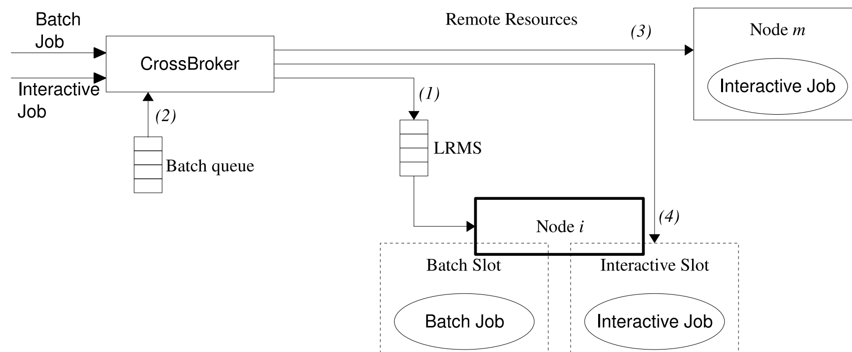


Figure 3.14: Multi-programmed execution of jobs.

1. Sequential Batch Job Submission. This submission triggers the execution of an agent if there is either an available machine or space in the queues managed by the local scheduler. Once started, the agent will create two virtual slots on the Worker Node/s: one for batch jobs (batch-slot) and another for interactive jobs (interactive-slot). The batch job will start its execution on the virtual slot devoted to batch applications (batch-slot). This situation is illustrated by arrow (1) in Figure 3.14. Special care has to be taken if the agent is killed (by

the local scheduler, by failure of the machine it is running on, etc.). In this case, new agents will be submitted when possible.

2. Interactive Application Submission (exclusive access mode): The CrossBroker tries to allocate a free machine. If there are CPUs available that meet the job requirements, the job will be submitted to it without any agent, illustrated by arrow (4) in Figure 3.14.

3. Interactive Application Submission (shared access mode): CrossBroker first searches for machines with agents, using the available interactive virtual slots (interactive-slot). The job will be sent to one of the virtual slots, which immediately meets the job requirements, causing the batch job executing on the other virtual slot to lower its priority so as to benefit the interactive job. This is illustrated by arrow (4) in Figure 3.14. If no free interactive agents are found, CrossBroker searches for an idle machine and submits the agent and the application in a similar way to the case of a batch job.

If there are no idle machines or there is no space in the local scheduler's queues, batch applications are queued within the CrossBroker waiting for a machine to become idle. This situation is illustrated by arrow (2) in Figure 3.14. However, if there are not enough machines (with or without agents) to execute an interactive application, its submission will fail. An interactive application will never preempt another already-running interactive application. Although not shown in Figure 3.14, it is possible to have a combination of machines with and without agents for executing a parallel interactive application.

The agent-based mechanism improves resource availability for interactive jobs such that they will even be able to run under conditions of high Grid-resource occupancy. This has little impact on batch jobs that undergo some execution delay when sharing their CPU with interactive jobs. However, given the nature of batch jobs, this delay is not particularly problematic.

Parallel inter-cluster applications may also benefit from this multi-programming environment. Since the reservation of machines is not supported by most LRMSs, the grid scheduler submits the different job components to the CE queues and waits until all of them have nodes allocated in order to allow the application to run. During the time elapsed from the allocation of the first job component until the allocation of the last one, the machines are idle. By submitting our agent instead of the real application, the CrossBroker can take advantage of the machines that would be otherwise idle by back filling until the whole application has machines allocated.

Figure 3.15 depicts the execution of a parallel inter-cluster application with this mechanism. Once the resources that will be used for the execution have been selected, instead of the job components, an agent is submitted to the LRMS queue on each

site as illustrated by arrow (1). The Application Launcher waits until all the job components have an allocated machine in order to let the job run. The LRMS at each site starts the agent as shown by arrow (2) in the Figure. Once started, the agent creates two virtual slots on the Worker Node/s: one for batch jobs (batch-slot) and another for the inter-cluster job (intercluster-slot). The job component is started on the intercluster-slot and waits until the Application Launcher allows it to run. While the inter-cluster application is waiting for all the job components to be ready, the CrossBroker gains batch-slots and may start batch applications. Arrow (3) illustrates this situation. When all the job components contact the Application Launcher, the user application starts its execution and the batch jobs on the batch-slots can be suspended, cancelled, or run with lower priority, depending on the policy implemented by the CrossBroker scheduler.

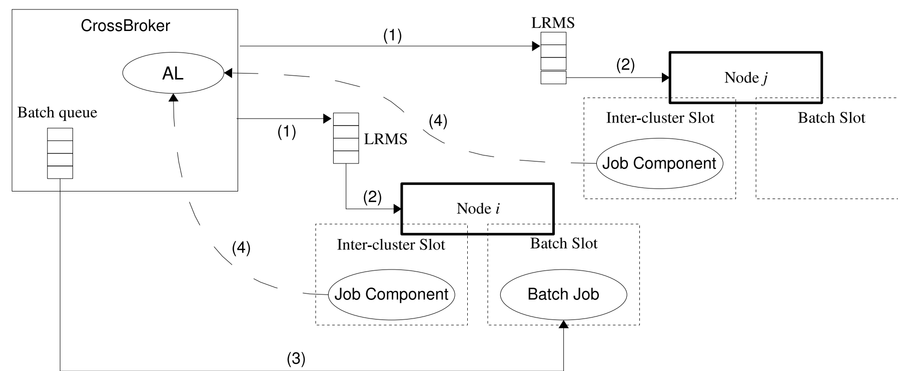


Figure 3.15: Multi-programmed execution of inter-cluster jobs.

3.5 Conclusions

A new architecture for running and managing parallel and interactive jobs in grid environments has been presented in this chapter. The execution environment for this architecture was presented. The model for jobs was presented next. This model defines three kinds of applications: serial, parallel, and workflow. A set of elements allow the actual execution of such applications on the grid. The **Application Launchers** manage the grid level start up of applications and the **Job Starters** are responsible for starting the applications at the resource level. The **Interactive Agents** support the interaction of the application with the user on-line. The combination of these elements allow the execution of applications using different parallel library implementations and different interactive channel forwarding mechanisms.

We have presented a job description language that allows the expression of the details of the job model proposed. It includes specific attributes for the definition of

parallel jobs and workflows, as well as the specification of interactivity features. The architecture of a Grid Resource Management System with specific mechanisms for interactive and parallel jobs was presented. The details of design and implementation of this system will be given in the next chapter.

CHAPTER 4

CrossBroker Design and Implementation

In this Chapter we describe the design and implementation of the CrossBroker. This implementation follows the architecture presented in Section 3.4. Figure 4.1 shows the main components of the CrossBroker:

Scheduling Agent. The entry point to the CrossBroker, in charge of making the scheduling decisions.

Resource Searcher. Discovers resources and matches user requirements with available resources.

Job Execution Components. The Application Launcher, Job Starter, and Interactive Agent manage the actual execution of the jobs on the remote resources.

In the implementation of the CrossBroker, we have tried to take advantage of the currently available development in the area. Each of the modules is described in detail in the following sections. Additionally, we present some examples of real applications that make use of the CrossBroker.

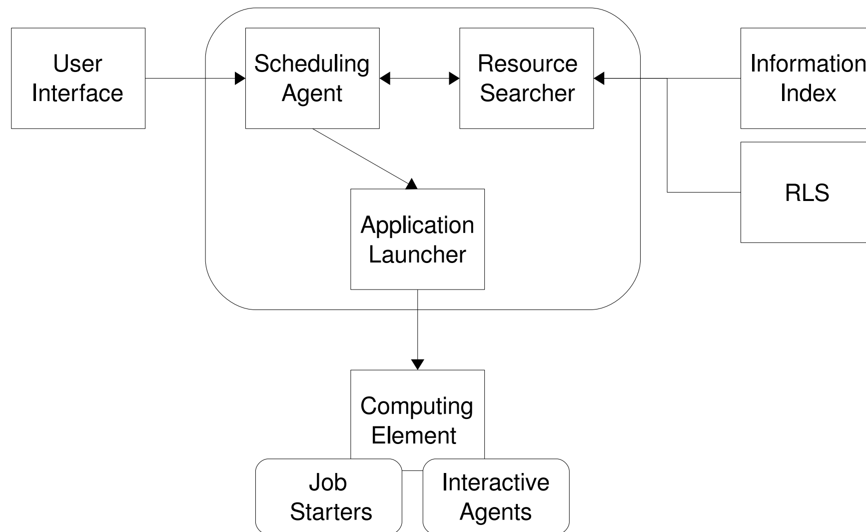


Figure 4.1: CrossBroker Architecture

4.1 Scheduling Agent

The *Scheduling Agent* (SA) is the entry point to the CrossBroker. It receives user jobs and decides where to run those jobs following one scheduling policy. In order to obtain a list of available resources, it uses the Resource Searcher. The contents of this list depend on the type of job, the job's requirements, and the current status of the resources.

Figure 4.2 shows the internal design of the Scheduling Agent. The *User Access Module* pre-processes the job and provides support for external tools that handle complex jobs such as workflows or parameter sweep applications. Once a batch job has been pre-processed, it enters a queue of pending requests. Interactive jobs go directly to the *Scheduler* module. This module applies the scheduling policies and selects the resource or set of resources for the job. In order to obtain a list of available resources it uses the *Resource Searcher*. The CrossBroker multiprogramming mechanism is managed by the *Glidein Monitor*.

The Scheduling Agent uses a persistent storage system called *Logging and Bookkeeping* (L&B) [95]. This service, taken from gLite, keeps information about the job life cycle and the different events that have occurred during the scheduling phase.

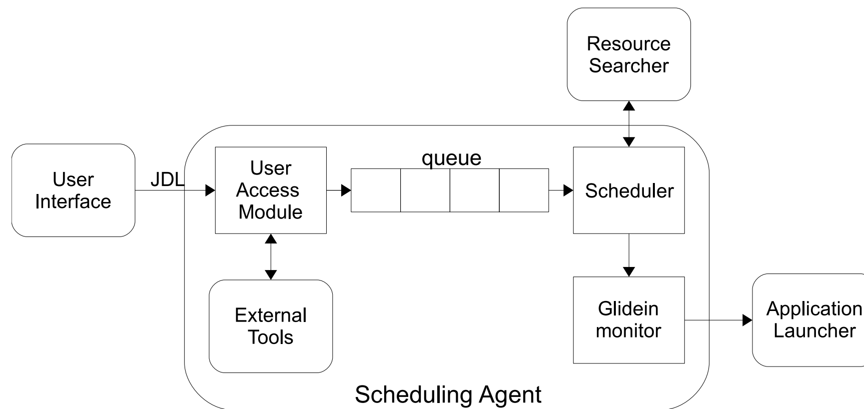


Figure 4.2: Structure of the Scheduling Agent

4.1.1 User Access Module

The User Access Module is the entry point to the CrossBroker. Its main tasks are:

- To establish and handle communication between the user and the CrossBroker,
- To receive user commands, checking each for its integrity and correctness,
- To invoke the appropriate modules in order to process the user request, and,
- To send back the responses to commands sent by the user.

Every user request is handled independently using threads. Moreover, all connections use Globus security mechanisms by using the proxy files associated with every request. Each user can only act upon his own jobs.

Submission Helpers

The User Access Module includes a *Submission Helper* interface. This interface allows the use of external tools for job submission. A Submission Helper can be registered for each job type and pre-processes the job. Currently, in CrossBroker we have included support for workflows using a Submission Helper for Condor DAGMan [96]. DAGMan can be treated as an iterator on a workflow, whose main purpose is to navigate through the graph (node by node), determine which nodes have their dependencies satisfied, and follow the execution of corresponding jobs.

The Workflow Submission Helper parses workflow jobs and creates an appropriate DAG description following the DAGMan syntax. A DAGMan process is started for

each workflow submitted. Hence if there is more than one DAG to execute, a separate DAGMan process is initiated for each workflow. Each node of the DAG managed by DAGMan is a job that will be handled by the Scheduler module. A set of steps operated on each node in the workflow:

1. Initial phase – preparing all necessary information for the node execution. The job is enqueued in the Scheduling Agent queue and there is a search for a suitable resource on which to run the job. If no resources are found, the node is marked as failed, which implies the end of the node's execution. A failed node can be automatically retried according to the `node_retry_count` value in the JDL.

2. Job execution on the remote site by using the appropriate Application Launcher.

3. Final phase – checking the job's return code. If the job executed successfully, it is marked as Done. Otherwise, the return value is compared to the attribute `app_exit_code`. If the return value is one of the values specified by the user, the job is not retried. In any other case the job is marked as failed and is retried according to the `node_retry_count` value.

The Workflow Submission Helper also supports the function of submitting a failed workflow and executing only those nodes that have not yet been successfully executed. This workflow is automatically produced by the system when one or more nodes in the workflow has resulted in failure, making the application execution impossible to finish. If any node in the workflow fails, the remainder of the DAG is continued until no more forward progress can be made, due to the workflow's dependencies. At this point, a file called a Rescue DAG is produced, which is given back to the user. Such a DAG is the same as the original JDL file, but is annotated with an indication of successfully completed nodes using the `status = done` attribute in the description file. If the Rescue DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be reexecuted.

User Commands

The User Access Module accepts a set of predefined commands described below:

- *job-submit*: submission of jobs. The job description is parsed and placed in the Scheduler queue. This process generates a unique identifier for the job, which can be later used to check the job status, cancel the job, or obtain the job's output.

- *job-cancel*: cancels a job or workflow.
- *job-list-match*: obtains a list of available resources that meet a job requirements. The Resource Searcher is contacted directly.
- *job-get-output*: receives the output files of an already completed job. These files must be specified in the `OutputSandbox` attribute of the JDL.
- *job-get-status*: checks the status of a previously submitted job. It uses the job identifier to query the current status or history of a job. The status for all jobs is kept in the *Logging & Bookkeeping* service.

4.1.2 Scheduler

The Scheduler receives the job requests and assigns a resource or set of resources for the job's execution. It uses the Resource Searcher module to fetch a list of available resources and sorts the list according to the user's preferences (specified by the `Rank` attribute in the JDL) and the Scheduler's own policies. These policies can be changed by using plugins to the scheduler. The policy plugins receives a list of resources and returns an ordered list of resources. The job will be executed on the first resource of the list. In case of failure, the subsequent list element will be tried.

In the current implementation, the policy takes into account the type of the job in order to make a decision:

Intra-Cluster and Normal jobs. Sites closer to the files requested by the job are selected first, hence access time to those files is minimized. Moreover, the queue lengths at the sites are checked in order to avoid overloading the sites. A Best Fit policy — the site with smallest difference between the number of CPUs requested and the number of CPUs available at the resource is selected first — is used in order to sort the list of available resources.

Inter-Cluster jobs. Besides checking the queue status, inter-cluster jobs are placed by trying to minimize the number of different sites used. The applied policy is:

- Sets with the smaller number of unique CEs are selected first. By using this policy, fewer sites are used, and hence high latency links are avoided for application execution.
- If there is more than one set with the same number of CEs, the more highly ranked set will be selected first.

Interactive jobs should be executed as soon as possible in order to minimize the response time. Hence, such jobs are never sent to sites without free resources currently

available to run the job. The resources selected must have as many free CPUs as requested by the job. If there are no sites with sufficient CPUs, the multi-programming mechanism is used and the interactive job is submitted to an interactive virtual slot (or a set of them). In any other cases the job is aborted immediately, so the user can decide wether to resubmit or wait.

4.1.3 Glidein Monitor

Once the Scheduler has selected the resource (or set of resources) on which to run the job, the Glidein Monitor enables as necessary the use of the multi-programming environment. This multi-programming scheme takes advantage of the Condor Glide-In mechanism, and is based on the transparent submission of job agents for jobs submitted by the user.

Whenever a user job activates the multi-programming environment, two jobs are passed to the Application Launcher:

1. A special "Glidein" job that will run on the remote resource selected by the Scheduler module. This job downloads Condor daemons onto the Worker Node and executes them with the appropriate configuration for appearing as a resource within a Condor Pool in the CrossBroker.
2. The user's job, that is submitted to the local Condor Pool, so it can make use of the resources created by the Glidein job.

The remote resource joins the pool as two Condor slots (`slot1` and `slot2` in Condor terminology), which are then available for running jobs sent from the CrossBroker. Once this environment is ready to accept jobs, it will appear in the list of Condor resources and will execute jobs taking into account the following considerations:

- Batch jobs can only be run in `slot1`.
- Interactive (or parallel inter-cluster) jobs can only be run on `slot2`.
- The job that originated the execution of the Glidein has precedence over any other jobs.
- Only jobs sent by users of the same VO as the originating job can be run.
- Batch jobs' priority can be changed during execution.
- When no jobs run on the resources for a configurable period of time, the Glidein exits, freeing the resources.

The Glidein is activated in two cases: batch sequential jobs and batch inter-cluster parallel jobs, i.e. batch (sequential or parallel inter-cluster) jobs will be submitted along with an agent in charge of creating the virtual slots. Interactive jobs have response time constraints, so introducing the extra overhead of the Glidein is not viable. Parallel jobs should never be preempted on a per process basis, or time-outs in communication nay cause the application to be killed. For this reason, parallel batch intra-cluster jobs do not activate the Glidein mechanism. In the case of inter-cluster jobs, as explained in Section 3.4.1, the different job components run within the inter-cluster slot (with maximum priority) and create batch slots that can be used for low priority jobs while the Application Launcher waits for all the components to be ready for execution.

Security and the Glidein mechanism

As the Glidein mechanism does not use any special privileges on the Worker Nodes, every process is run from a local user account without administrative permissions. This allows the Glidein to be easily deployed on the sites, but it is also a potential security risk. Both the Condor daemons and the user jobs on the virtual slots run within the same user account. Hence, a malicious program could kill the job executing on the other virtual slot or even kill the Condor daemons. Additionally, this malicious program may access to the files owned by the other job.

The use of full featured virtual machines does not have this security problems. The virtual machine completely isolates the running environment, by running a separate operating system image. However, special priviledges are needed in most of the virtual machines technologies rendering the solution unusable in grid environments where the jobs are run as unpriviledged users.

Ideally, the operating system should provide mechanisms for 'jailing' the processes. However, most systems implement a user based security that is not enough for our purposes, because both the jobs and Condor daemons are run within the same user account. We propose the use of Identity Boxing techniques [97]. They provide a secure system-call interposition agent; this agent provides fine-grained control over the operations allowed to the applications run in the environment without special privileges. An identity box is a well-defined execution space in which all processes and resources are associated with an external identity that need not have any relationship to the set of local accounts. A single Unix account may be used to securely manage several identity boxes simultaneously, thus eliminating the need for services to run as root merely to change identities. An available identity boxing implementation based on Parrot [98] intercepts and modifies system calls through the ptrace debugging interface. This provides secure identity boxing at the user-level on arbitrary unmodified programs.

In our system we use Identity Boxing to control all file system accesses, we restrict file access for user application to a specific directory. We control processes creation by forcing those processes to be under the control of the identity box. We also control the kill system call by only permitting the sending of signals to processes within the application process tree. In this manner we avoid malicious software kill Condor processes or other applications running on the virtual slots. The use of this technique has a non-negligible, but not-unreasonable overhead [97]. Scientific applications are slowed down by only 0.7 - 6.5 percent. An interactive application such as make slows by 35 percent, because it makes extensive use of small metadata operations such as stat.

Multiprogramming on sites without an inbound connection

Our multiprogramming mechanism relies heavily on Condor features and daemons in order to create the virtual slots on the remote resources and to start the jobs on such resources. Due to Condor's pattern of network communication, the multiprogramming schema proposed requires a direct inbound connection from the CrossBroker to the Worker Nodes. This is a potential security risk, and most sites follow a "deny all inbound connections; permit some outbound connection" policy within their firewalls. Moreover, Worker Nodes usually have private IP addresses, which render impossible any kind of connection from outside the site.

A new firewall/NAT traversal technique known as Generic Connection Brokering (GCB) [99] has been introduced in Condor. It consists of daemon processes and a communication library that is already linked to the Condor binaries. GCB enables connections into networks behind a firewall/NAT by reversing the direction of the connections. The daemon processes should be installed on a machine accessible from both the Worker Nodes and the CrossBroker. The Computing Element has such capabilities. We have created a special package that includes GCB and configuration tools that can be installed on the CE of each site, allowing the multiprogramming mechanism to work without firewall modifications. Figure 4.3 depicts the connection pattern with the GCB mechanism enabled. The red arrow shows the typical communication pattern of Condor, while the black arrows identify the pattern when GCB is enabled.

4.2 Resource Searcher

The main duty of the *Resource Searcher* (RS) is to perform matchmaking between job needs and available resources. Figure 4.4 shows the internal design of the Resource Searcher. The Matchmaker module receives a job description as input, and outputs a list of possible resources on which to execute the job. Available resources are stored

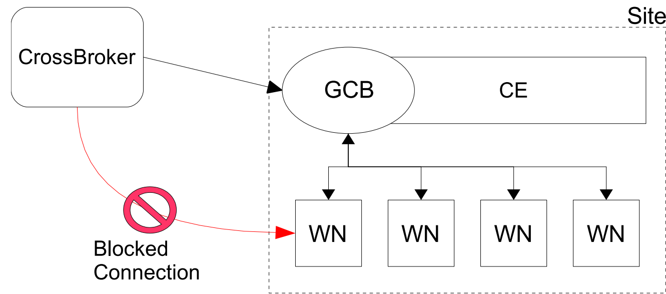


Figure 4.3: Use of GCB with glide-in

in a Resource Cache. The Resource Cache is updated at regular intervals by a set of *updater* modules. Currently, there are updater implementations for fetching data from R-GMA [33], BDII [100], or a Condor Collector [22] with information about the available Glideins. The information gathered from the different sources is merged to produce a list of resources without duplicates in ClassAd format.

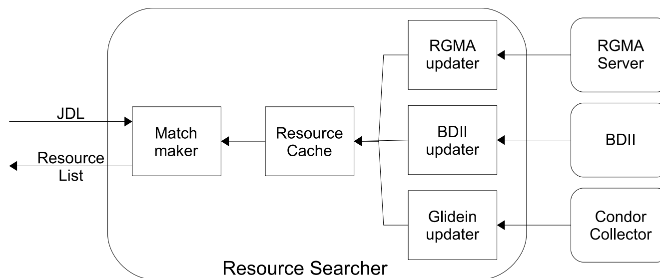


Figure 4.4: Structure of the Resource Searcher.

4.2.1 Resource Cache

The available information about the resources is a key point in grid environments. Outdated information may cause making improper scheduling decisions and consequent job failures, while too many updates may overload the system, rendering it unusable. In the CrossBroker we use a Resource Cache for information about available resources and it is updated at regular time intervals. These intervals are short enough to maintain up to date information, while maintaining a light system load.

Each resource in the cache is described as a ClassAd, identified with a unique name. The different attributes that compose the resource information follow the Glue Schema [31] (see Section 3.1.) The information stored includes both static attributes

(such as number of CPUs, system architecture, LRMS, and available software) and dynamic attributes (such as system load, number of jobs running or waiting, number of free CPUs, and estimated response times).

The Resource Cache has a mechanism to assure exclusive temporal access to resources. Resource selection may occur concurrently for several jobs that arrive simultaneously. This exclusive access mechanism guarantees that one resource is only matched and allocated to one job for a specific time interval. This mechanism eliminates the lack of the most current information about the status of grid resources and it also helps to avoid situations where the same available resources are assigned to more than one resource petition, thus creating a deadlock for MPI jobs.

The information from the available information repositories is fetched by the *updaters*. These modules fetch the information, adapt it to the ClassAd format, and merge the latest information with the existing information in the Resource Cache. They are triggered asynchronously, every two minutes by default. We have found empirically that this value is adequate to keep updated information about the resources without introducing incorrect scheduling decisions. The CrossBroker provides an API for developing such updaters. Currently there are three updaters implemented with that API:

MDS updater. This is the main source of information in the CrossBroker, it does a two-level query. First, it gathers information from the central Information Index which contains information about the resources available. Then it gathers information about each resource listed in the Information Index by directly connecting to the resource. Although this process can be time costly given the number of resources available, it occurs independent of the scheduling process. This updater uses LDAP for fetching resource information.

RGMA updater. This updater fetches resource information from an RGMA server. It is not used in the current implementation, since the resource information can be fetched from the MDS.

Condor Collector updater. Information about Glideins in the local Condor Pool of the CrossBroker is fetched with this updater. It uses the `condor_status` command to query the Condor Collector daemon. Since all the information gathered by this updater is available locally, it can be invoked directly by the Scheduling Agent in order to obtain the most recent information about the available Glideins for interactive jobs.

Although the information described by Glue Schema is quite complete, it is not yet fully implemented in the available information systems. This is problematic for of parallel jobs, where site limitations can make jobs fail without apparent reason. As a temporal solution, the CrossBroker can use the `RuntimeEnvironment` attribute

of the Glue Schema to specify site characteristics and limitations. Originally the attribute allowed the specification of the installed software available at the resources. It appears once for each piece of software, with no format or length constraints. We decided to include here all the attributes needed for parallel jobs at the int.eu.grid project level, in addition to any other information that the site administrator may like to include about software. The CrossBroker looks in the attribute for specific tags that identify MPI implementations at the site. The tags are configurable at the CrossBroker level. Additionally, some sites have restrictions on concurrent use of the number of nodes that can be used for a given job. This restriction is not properly reflected in the current Glue Schema implementation used in the CE, thus creating incorrect matchmaking for MPI jobs. In order to avoid this situation, we use special tags with the following pattern:

```
PolicyMaxSlotsPerJobs_<QueueName>=<MaxNumber>
```

where `<QueueName>` is the name of the queue with restrictions and `<MaxNumber>` is maximum number of nodes that can be used simultaneously by an application. If this tag is not defined for a queue, the CrossBroker assumes that the queue can execute jobs using the maximum number of CPUs available at the site. Again, the format of this tag is configurable at the CrossBroker level.

Separate from previously detailed sources of information, there is a specific updater that can locate replica of files in the grid environment. This updater is triggered by the Scheduling Agent when a job requires a file. The updater receives a list of files and queries the RLS [101] using Web Services to return a list of remote sites that have the files available. That information is used by the scheduling process in the selection of sites.

4.2.2 Matchmaking

The matchmaking process carried out by the Resource Searcher is implemented with the Condor ClassAd library. With this library, jobs and resources are expressed as ClassAds; two ClassAd match if each the attributes evaluates to true in the context of the other ClassAd. Because the ClassAd language and the ClassAd matchmaker were designed for selecting a single machine on which to run a job, we have added several extensions, applied when a job requires multiple resources (i.e. multiple CEs in our environment terminology).

Prior to the matchmaking process, the matchmaker adds requirements to the job description some requirements to ensure the selection of appropriate resources for the job. These requirements include:

Security Requirements. The users have permission to execute on a restricted set

of resources. With the inclusion of the user credentials in the job, it is possible to discern between allowed and forbidden sites.

Software Requirements. Some jobs need specific software installed at the remote sites. This is especially important for parallel jobs, where a specific MPI implementation must be installed for proper job execution. The matchmaker module ensures that the sites have the correct libraries available at run time.

CPU Requirements. Intra-cluster parallel jobs need more that one CPU for its execution at one site. For this case, the matchmaker ensures that the sites to have at least the number of CPUs requested. Inter-cluster jobs use the set-matching algorithm described below.

During matchmaking, each of the resources available in the Resource Cache is compared to the job requirements. If they match, then the resource is put into a list of matching resources and the rank of the resource for the given job is calculated. Once all the resources have been checked, the list is returned to the Scheduling Agent that will select one for job submission.

Set-matching

Inter-cluster parallel applications use a specific matchmaking algorithm that is able to match a single job to a set of resources. This algorithm is called *set-matching*.

In the set-matching algorithm, a successful match occurs when a ClassAd set (a group of CE's ClassAds) satisfies all constraints as set by a ClassAd (the Job). First, the job is places constraints on the collective properties of an entire group of CEs ClassAd (e.g., the total number of free CPUs must be greater or equal than the minimum number of CPUs required by the job). Second, other attributes of the job ClassAd are used to place constraints on the individual properties of each CE ClassAd (e.g., the OS version of each CE must be Linux 2.4). The selection of resources is carried out in accordance with the following steps:

1. Obtain a list of single CEs that fulfill all job requirements, when referring only to required individual characteristics. Currently, these are the requirements specified in the `requirements` attribute of the file describing the job using the JDL, as well as the ones added by the matchmaker module. This step constitutes a pre-selection phase that generates a subset of the total resources suitable for executing the job request in terms of several characteristics such as processor architecture, OS, etc.

2. Fulfill collective requirement for groups of CEs from the subset of resources determined in step 1. For example, an attempt is made to fulfill the total

number of CPUs required by a job by aggregating individual CEs. In the case of the number of CPUs required by the job, for instance, the Resource Searcher aggregates CEs to guarantee that the total number of free CPUs in the groups of CEs is larger than or equal to the total number of requested CPUs, as described in the JDL file.

Our search procedure is not exhaustive, as it does not compute the power set of all CEs. In particular, this means that our search algorithm does not consider solutions in which one subset of the CEs has already been included in a previous group. Consider the example of a list of resources composed by (CE1, CE2, CE3, CE4), where the following sets meet the collective requirements: {CE2}, {CE1, CE3}, and {CE1, CE4}. The set-matching algorithm would not consider the {CE1, CE3, CE4}, since {CE1, CE3} is already a valid solution. Additionally, a maximum number of elements per set can be defined in the algorithm, imposing an upper limit on the number of sets to be evaluated.

4.3 Job Execution

The actual execution of job on the remote sites is handled by the Application Launcher, Job Starter and Interactive Agents modules. For every job an Application Launcher submits a Job Starter to the remote resource. This Job Starter will in turn download the job files and start the application and Interactive Agent if needed. Figure 4.5 shows the interaction among the job execution components. The Scheduling Agent selects the resource or list of resources that will be used for the execution of the job. With this list, the Application Launcher uses Condor-G as its interface to the Computing Element and submits the Job Starter and Interactive Agent needed to run the job. The user's application is executed under this environment, contacting the Application Launcher for synchronization and final configuration steps.

4.3.1 Application Launcher

Each kind of job has a specialized Application Launcher associated with it. In the current CrossBroker implementation, the various Application Launchers use Condor-G [36] job management mechanisms for submission of jobs on remote resources. Condor-G gives a consistent interface for submitting jobs to grid resources using Globus resource access and security protocols (GRAM and GSI respectively.); Condor-G also guarantees fault tolerance and exactly once execution semantics thanks to a persistent (crash proof) queue of jobs, used as a persistent database storing information concerning active jobs with a two-phase commit protocol for job management operations. Condor-G interoperates with other systems supporting, besides Globus

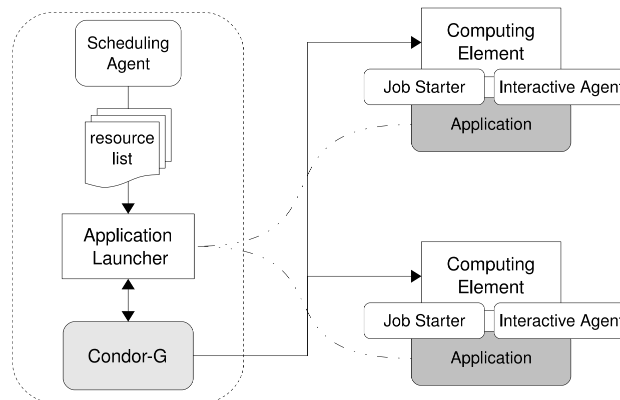


Figure 4.5: Job Execution Components.

version 2, other Globus versions, Unicore, NorduGrid, and submission to other batch systems as PBS and LSF. All those systems are potentially supported by the CrossBroker by using Condor-G as an interface to the grid. Currently, there are two kinds of Application Launcher in the CrossBroker:

Single-site AL. This Application Launcher handles all the jobs that are executed on only one site. The Application Launcher receives the job description along with the resource from the Scheduling Agent and submits the application to the remote site using Condor-G. A single AL process manages all the single-site jobs. In case of failures, the AL can recover the system status by using the persistent queue of Condor-G. Intra-cluster parallel jobs also use this Application Launcher.

Inter-cluster ALs. Inter-cluster applications have dedicated Application Launchers that manage the job details for start up and synchronization. They receive a list of resources along with the job description. Application using the MPICH-G2 implementation of the MPI standard are submitted using the MPICH-G2 Application Launcher, while the PACX-MPI AL manages the PACX-MPI applications. The inter-cluster ALs use Condor-G as the single site AL. Moreover, in order to provide extra reliability, these ALs themselves are submitted to the local Condor-G queue. Hence, in case of an AL crash, Condor-G will resubmit the AL and the AL may continue the execution of the job.

In order to support new parallel application implementations, the CrossBroker provides a C++ API to define the details of the protocol that depend on the different applications and communication libraries. The API allows the definition of:

- Submission method. The submission method defines how to start the subjobs

on the remote resources, and the Application Launcher provides a Condor-G submission method, ready to use.

- Synchronization method. Each MPI implementation uses different synchronization methods for the start up of application. MPICH-G2 uses Globus communication services and barriers, while PACX-MPI provides *startup-server* for this purpose. The Application Launcher API allows the developer to define the synchronization procedure according to the application implementation.
- Monitoring method. Once the application is up and running, it must be monitored to check that it finishes its execution properly. This monitoring also includes error handling, for when problems arise during the execution. A Condor-G monitoring method is also provided.

MPICH-G2 Application Launcher. Once the Scheduler Agent (SA) detects that an MPICH-G2 application is submitted, an MPICH-G2 Application Launcher (MPICHG2-AL) is submitted to the local Condor-G queue in the Crossbroker. Figure 4.6 depicts how the execution over multiple sites is performed. In this example scenario, N subjobs constitute an MPICH-G2 application. These subjobs will be executed on different sites. For the sake of simplicity, Figure 4.6 only shows 2 sites. This MPICHG2-AL coallocates the different subjobs belonging to the parallel application, following a two-step commit protocol:

- In the first step, all the subjobs are submitted through Condor-G. The type (A) arrows show the submission of subjobs to the remote sites. Condor-G uses a GASS server to stage the Job Starter to the remote worker nodes which will handle the download of executable files and to bring the output files back to the submitting machine. This is shown by the type (B) arrows.
- A second step guarantees that all the subjobs have a machine to be executed on, and that the subjobs have executed the MPI Init call. This MPICH-G2 call invokes DUROC [55], and synchronization is achieved by a barrier released by the MPICHG2-AL. After synchronization, the subjobs will be allowed to run. Once the subjobs are executing on the worker node machines, the MPICHG2-AL monitors their execution and writes an application global log file, providing complete information about the subjobs' execution. This monitoring is represented by the type (C) arrows in Figure 4.6, and constitutes the key point for providing reliable execution of the applications and robustness.

PACX-MPI Application Launcher. PACX-MPI jobs have a PACX-MPI Application Launcher (PACX-AL) that works the same way the MPICHG2-AL does, but using the PACX startup-server [77] instead of the DUROC API. The PACX-AL forks

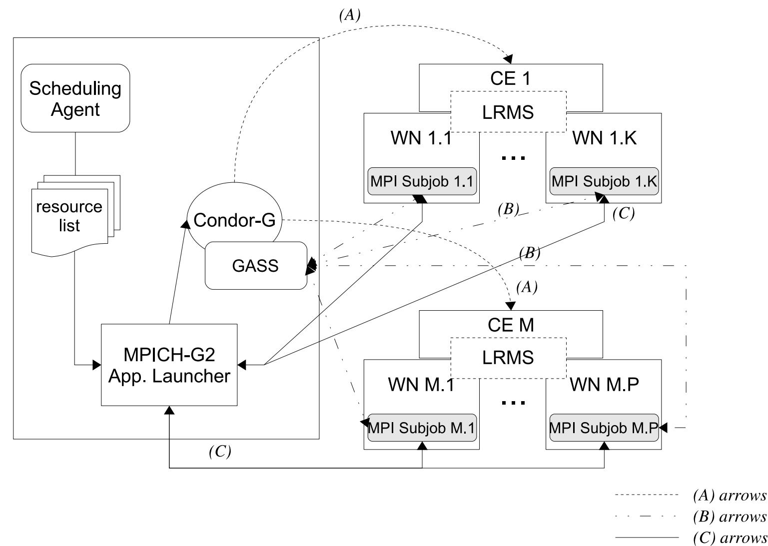


Figure 4.6: MPICH-G2 execution on multiple sites

a startup-server process that waits until the jobs have executed the MPI Init call. The exit code of this process will determine if the start up was or was not successful. Figure 4.7 depicts the execution of a PACX-MPI job over two different sites. First, the PACX startup server is forked on the CrossBroker machine. The information about this server is passed along with the subjobs and submitted using Condor-G to the remote sites as shown by the type (A) arrows. Similar to the MPICHG2-AL, the GASS server is used to stage the Job Starter to the remote worker. This is shown by the type (B) arrows. When the subjobs contact the startup-server (type (C) arrow in the Figure), this process ends and the exit code is checked by the PACX-AL. If no errors are detected, the job is monitored until the end of its execution.

Plain applications. If the user application does not fit any of the job types provided by the CrossBroker, the `plain` subtype may be used. The CrossBroker may use a specific AL that handles the submission of those jobs as specified by the JDL attribute `ApplicationLauncher`. This Application Launcher must be available at the CrossBroker, and it must be signed by a trusted authority, certifying that it conforms with the expected behavior for an Application Launcher. If not specified, the job is treated as a single site job and the corresponding AL is used.

Either if the application ends correctly or if there is a problem in the execution of the application, the AL records this in a log file, The log file will be checked by the Scheduling Agent, which will take the correct action, in accordance with that information. This provides a reliable once-only execution of the application without

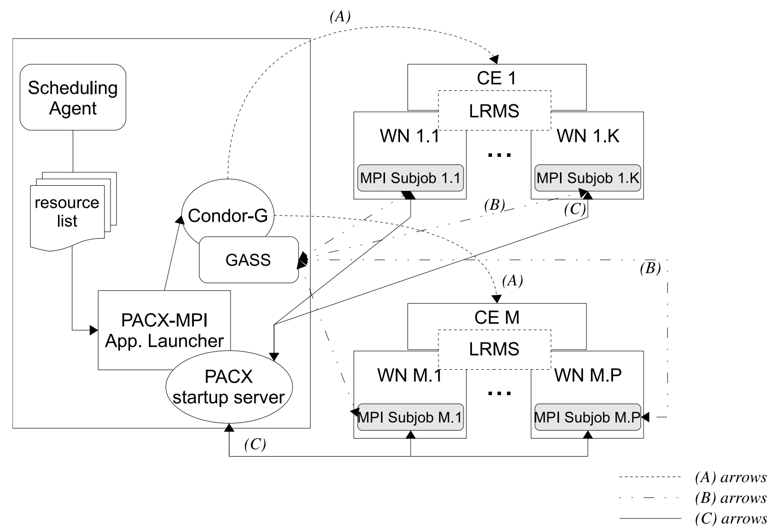


Figure 4.7: PACX-MPI execution on multiple sites

user intervention. The events logged by the Application Launcher are classified:

- Scheduled event: the application is submitted and is waiting for the start of all the subjobs.
- Running event: the synchronization protocol has finished correctly and the application is running.
- Done event: the application has ended its execution correctly.
- Error event: once an error has appeared, the AL logs the type of error and the remote site at which the error originated.

The problems detected can occur at different moments, and can be either temporary or permanent problems. Table 4.1 lists the problems that can appear during the execution of an application and the corresponding corrective action taken by the CrossBroker.

4.3.2 Job Starter

The Job Starter is the module responsible for starting the user application on remote resources. The CrossBroker submits a Job Starter instead of the user application for execution on the grid, and this fetches all the needed files and creates the appropriate job environment for execution. The basic Job Starter in the CrossBroker is written as

Detected situation	Action taken
One of the subjobs did not execute due to a resource down error.	Mark resource as down, repeat the matchmaking process.
One of the subjobs did not execute due to a Globus error.	Mark the resource as temporarily not working, repeat the matchmaking process.
One of the subjobs finished before the application started.	Retry execution with the same resources, if situation repeats, it could be a firewall issue, so repeat the matchmaking process.
The AL crashes.	Submit another AL instance. If the job has already passed the synchronization phase, the new AL instance will monitor it. In any other case, the job will be resubmitted to begin again.
Abnormal termination of job during execution	If no resource errors are found, the job is aborted and the user is notified.

Table 4.1: Errors Detected by the AL

a shell script, hence it can run on almost any system without major modifications. Moreover, this allows easy modification when incorporating new features into the system. All job-dependent variables are set as environment variables that the Job Starter can use. Table 4.2 lists the most important variables. These variables define the job environment, the different files that will be downloaded to the Worker Node (and from where to fetch them), the application name, its arguments, and the input, output and error files that it will use. Additionally, they define the interactive agent and the arguments that it will use. There is also a set of variables that specify the number of nodes and the index of the job component of the application.

Most sequential applications run without problems using the basic Job Starter provided. However, parallel applications have specific start-up mechanisms that can not be handled by the CrossBroker without introducing too many low level details about the applications and the Local Resource Management Systems. Moreover, these implementation details change more often that not, making it difficult to keep pace with them at the broker level. For these reasons, the CrossBroker uses external Job Starters for parallel applications.

The user can provide his own Job Starter by using the JDL attributes `JobStarter` and `JobStarterArguments`. This Job Starter is invoked at the Worker Node once the input files of the application are downloaded. In addition, the user can let the CrossBroker automatically handle the start up of MPI jobs using MPI-Start as a Job Starter, without worrying about any details. MPI-Start was developed in the int.eu.grid project framework as a Job Starter that the CrossBroker can use for

Variable	Meaning
`__jobid`	Unique job identifier.
`__environment`	Comma separated list of user defined environment variables.
`__input_file`	List of input files needed by the application.
`__input_base_url`	Base URL location for the input files.
`__job`	User's executable.
`__arguments`	Arguments for the user application.
`__standard_input`	File used as standard input for the application.
`__standard_output`	File used as standard output for the application.
`__standard_error`	File used as standard error for the application.
`__interactive`	1 if the job is interactive, 0 otherwise.
`__interactive_agent`	Executable used as Interactive Agent.
`__interactive_agent_args`	Arguments passed to the Interactive Agent.
`__nodes`	Number of nodes the application will use at the current site.
`__subjob_index`	If part of a inter-cluster application, index of the current subjob.

Table 4.2: Job Starter environment variables.

starting applications. It is able to start Open MPI [51], MPICH-P4 [44], PACX-MPI [77], and MPICH-G2 [45] on clusters using PBS [20] and SGE [23] resource management systems.

MPI-Start

The MPI Forum [73] gives recommendations on the mechanism for running MPI applications in the MPI-2 specification [43]: *mpirun* should be a portable and standardized script, while *mpiexec* is implementation specific. However, the different MPI vendors already were using *mpirun* in a non-portable and non-standardized way. In some implementations both *mpiexec* and *mpirun* are identical, while other implementations do not support one of them. Another problem is the distribution of the binaries and input files to all the nodes involved in a single execution: some clusters have a shared file system, while others need a mechanism to copy files from the head node to the rest of the nodes.

MPI-Start is a set of scripts that allow the execution of MPI programs on clusters. The main advantage of MPI-Start is the ability to detect and use site-specific configuration features – such as the batch scheduler and the file system at the site. Also, on site MPI implementations on a site are supported. MPI-Start has core functionality which is always executed and uses the available framework at different stages

of execution. MPI-Start in its current version has three different frameworks:

- Scheduler framework. Every plugin for this framework provides support for different Local Resource Management Systems (LRMS). They must detect the availability of a given scheduler and generate a list of machines that will be used for executing the application. The supported LRMSs are SGE, PBS and LSF.
- MPI framework. This plugins set the special parameters that are used to start the MPI implementation. It is not automatically detected, hence the Cross-Broker must explicitly specify which MPI flavour will be used to execute the application. There are plugins implemented for Open MPI, MPICH (including MPICH-G2), MPICH2, LAM-MPI and PACX-MPI.
- File distribution framework. The plugins for this framework handle file distribution among the machines involved in an execution. Different methods are supported, including shared file system use, passwordless scp, and an MPI implementation for copying files.

All of those frameworks use well-defined interfaces, which allow the various plugins to be loaded and used for each framework. The interface between the CrossBroker and MPI-Start is a set of environment variables listed in Table 4.3. When an MPI job is submitted and no Job Starter is specified, the CrossBroker invokes MPI-start setting the variables to the appropriate values for the job. These variables specify the application binary that will be executed, its arguments, and the interactive agent that will be used for I/O forwarding. They also define the MPI characteristics such as the MPI implementation, and in the case of inter-cluster execution, the index of the subjob, the node used as relay, and the machine where the startup-server is running.

Additionally, the user can modify the behavior of MPI-Start using the `Environment` attribute of the JDL, specifying the `I2G_MPI_PRE_RUN_HOOK` and `I2G_MPI_POST_RUN_HOOK` variables. Any other `I2G_MPI_*` variables defined by the user will be overridden by the CrossBroker in order to avoid misuse.

Although MPI-Start may be installed on the resources, when a MPI application is submitted, the CrossBroker checks at runtime if it is available locally. If not installed, it is downloaded automatically to the Worker Node.

4.3.3 Interactive Agents

The jobs submitted to the CrossBroker can use the interactivity mechanisms to interact with the application on-line. These mechanisms follow the architecture proposed

Variable	Meaning
I2G_MPI_APPLICATION	The application binary to execute.
I2G_MPI_APPLICATION_ARGS	The command line parameters for the application.
I2G_MPI_TYPE	The name of the MPI implementation to use.
I2G_MPI_PRECOMMAND	The Interactive Agent for interactive I/O forwarding.
I2G_MPI_FLAVOUR	Specifies which "sub-mpi" to use. In the case of a PACX-MPI job, this variable specifies the local MPI implementation to use.
I2G_MPI_JOB_NUMBER	If a MPI job runs across multiple clusters this variable specifies which sub-job should be started on this cluster. The values are 0, 1, 2,... In the case of an MPI job that runs only inside the local cluster this variable is always 0.
I2G_MPI_STARTUP_INFO	This variable provides additional information for the MPI program. In the case of PACX-MPI, this variable specifies the connection information to the startup server.
I2G_MPI_RELAY	This variable specifies an FQDN of a host that can be used as relay/proxy host. In the case of PACX-MPI on this host the additional 2 proxy processes will be started. It's required that this host has out-bound connectivity and is accessable via ssh.

Table 4.3: MPI-Start environment variables.

in Section 3.2.3: a split execution system where an Interactive Agent is in charge of sending the output and receiving input from the Worker Node to a Job Shadow on the user machine.

The Job Starter, using the `__interactive_agent` environment variable, starts the Interactive Agent once all the needed files are downloaded, and an appropriate environment is created for job execution. The Interactive Agent will, in turn, start the application and redirect its input and output. Currently the CrossBroker includes two Interactive Agents: one based on Condor Bypass [102], and the other based on glogin [94].

Figure 4.8 shows the general execution pattern for both agents. When an interactive job is submitted, the User Interface Command automatically starts an Interactive Shadow specifying a free available port for communication with the Agent. This port and the selected agent will be declared in the JDL with the `InteractiveAgent` and `InteractiveAgentArguments` without user intervention. The CrossBroker receives this job description and selects a resource for job execution. Once the job has Worker Nodes allocated for the execution, the Job Starter creates the Interactive Agent, which in turn starts the user application. Then, communication can be established between the Agent and Shadow and the application I/O will be forwarded from the User Interface to the Worker Node.

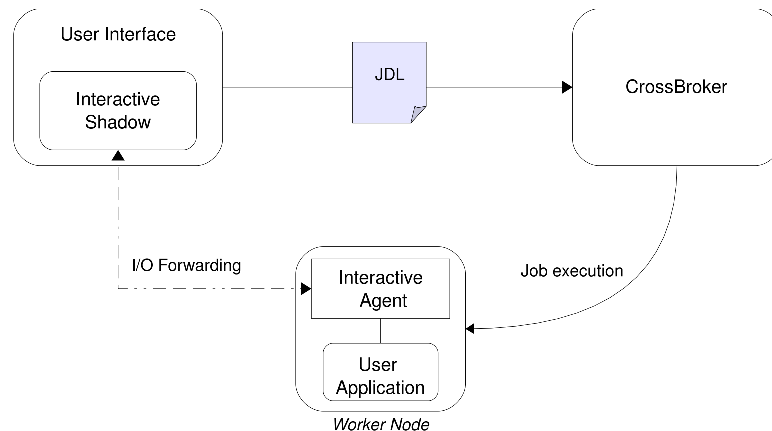


Figure 4.8: Interactive Agents in CrossBroker

Condor Bypass Agent

The Condor Bypass Agent obtains mostly-continuous I/O from remote programs running on an unreliable network. It is a split execution system composed of two

software components: an agent (Console Agent – CA) and a shadow (Job Shadow – JS). In CrossBroker the CA acts as an Interactive Agent and the JS as the Interactive Shadow described in Section 3.2.3.

In our case, the CA runs on a Worker Node, and it consists of a shared library that intercepts read and write operations for stdin, stdout, and stderr of the running job. When possible, the CA sends the output back to the JS (via RPC) as illustrated by Figure 4.9. If sending the output fails, the CA will instead write its output to the local disk. Regardless of why the I/O operation failed, the CA keeps the process running. At regular intervals, it tries the network connection again. If the connection succeeds, it transfers any buffered data to the JS, and then resume normal operation. The CA retries failed operations at regular intervals for a certain number of times, after which it gives up and kills the process. The number of retries and the number of seconds to pass between each retry are configurable.

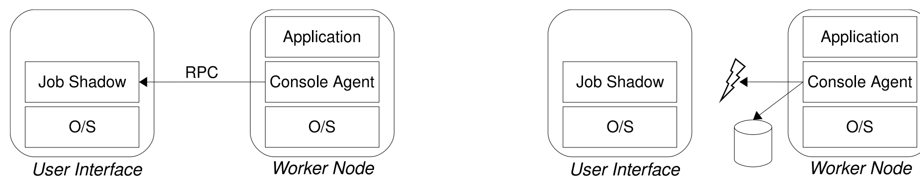


Figure 4.9: Condor Bypass Interactive Agent. Left: Normal Operation. Right: Operation in the case of network failure.

The start of the JS is handled automatically by the command line tools provided with the CrossBroker. First, JS is started to listen on the first free available port within the range of open ports for Globus communications. This port and the IP address of the user machine is sent to the CrossBroker in the `InteractiveAgentArguments` of the JDL which will be used later by the CA in the WN. It is worth noting that buffers have been included for both the submitting and executing machines to provide users with a genuine feeling of interactivity for text oriented applications. The output buffer associated with the JS process located on the submitting machine is flushed to the screen in 3 cases:

- When the output buffer on the submitting machine is full.
- When a timeout occurs. This situation corresponds to the case when users are requested to input data.
- When an “end of line” is encountered.

The input provided by users on the submitting machine is forwarded to the input buffers associated with each CA on the executing machines. The forwarding is produced when the “enter” key is hit. This flushing is intended to provide a readable

output for users, and a "natural" appearance. However, it is only useful for text-oriented applications.

i2glogin

i2glogin is a tool to enable interactive communication between a grid job and the user interface. It is a spinoff of the original glogin tool [94], and most of the features are available in both versions. The fundamental difference between the two tools is given by the job submission mechanism used. While glogin submits itself automatically to the grid, i2glogin must be submitted explicitly using the grid middleware.

For building an interactive tunnel, two instances of i2glogin are needed, one local instance at the user interface and a remote one on the grid. The local instance allocates a TCP-port and waits for the remote instance to establish the connection. Locally, i2glogin is started by user command line tools, while the i2glogin's grid instance is submitted by using the JDL file options. When started, the local i2glogin prints its connection parameter to standard output. This parameter is used by the command line tools in the JDL file, because the parameter must be used by the remote i2glogin.

The i2glogin tool not only provides remote forwarding of the input and output files of an application, it is able to offer different interactive services, such as pseudo terminals, TCP forwarding (including remote X11 displays), and grid-enabled virtual private networks (VPN). Moreover, as in the case of the Bypass Agent, all the communication is encrypted by Globus GSI tools.

The port forwarding and tunneling features of i2glogin allows the use of higher level tools such as GVid [52] on top of i2glogin. GVid is a grid-based video service which can be used to transmit the output of visualization processes running on the grid, to the user's desktop machine. Therefore the visualization can be executed as a remote grid job while the user is observing and interacting with the visualization from the local desktop machine. GVid captures the output of an arbitrary visualization based on OpenGL, X11, or VTK, and transmits it to the remote user as a video stream.

4.4 Example Applications

The CrossGrid and int.eu.grid projects identified a set of applications that benefit from the deployment of an infrastructure like the one proposed in each of the projects. These applications are characterized by making use of MPI for communication between processes and/or the necessities of interactive steering or visualization.

Plasma Simulation

The flagship application of int.eu.grid is a simulation and visualization of the behavior of plasma in nuclear fusion devices [103]. In this application the plasma is analyzed as a many-body system, composed of electrons and ions, orbiting inside the fusion device cavities. This approach has become feasible only in the past two years, because researchers can profit of large scale clusters at their own centers, and distributed computing infrastructures like the grid. Furthermore, the visualization of the plasma particles inside fusion devices is an interesting tool because the nature of the plasma makes it very difficult to obtain direct experimental measurements of real fusion devices.

Figure 4.10 shows the application running inside the Migrating Desktop. The application is a Master/Worker MPI application with intensive communication, that generates an OpenGL visualization transmitted to the user desktop with Gvid [52] and glogin.

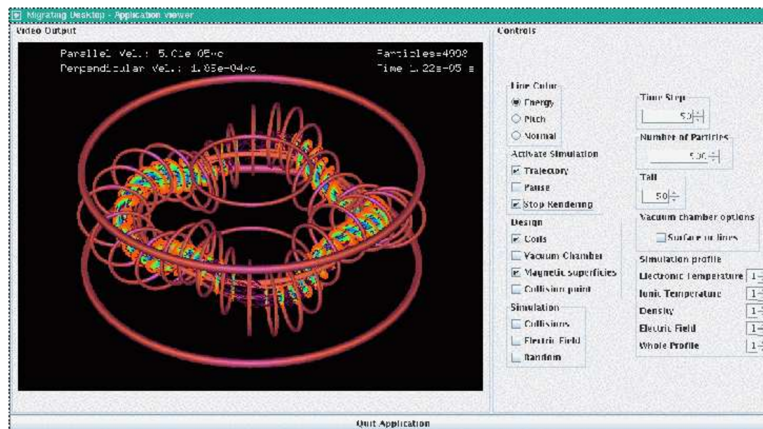


Figure 4.10: Plasma visualization application running on the Migrating Desktop

The JDL description of this application is given in Figure 4.11. It is an Open MPI parallel application that executed on 4 CPUs. The executable is linked with the GVid tool and receives two arguments that are used by this tool to generate the video output. The interactive agent used in this case is i2glogin, and its arguments specify the client machine and port (158.109.65.149, port 12345). The application uses a file for input data that contains the plasma device geometry and the initial characteristics of the plasma simulated. This input file and the executable are transferred to the Worker Node by the CrossBroker. The environment variable `POSIXLY_CORRECT` is needed for the proper functioning of Gvid.

```
JobType         = "Parallel";
SubJobType      = "openmpi";
NodeNumber      = 4;

Executable      = "run_fusion";
Arguments       = "-vid -client";

Interactive               = True;
InteractiveAgent          = "i2glogin";
InteractiveAgentArguments = "-r -p  12345:158.109.65.149 -c ";

StdOutput       = "std.out";
StdError        = "std.err";
OutputSandbox   = {"std.out","std.err"};
InputSandbox    = {"input.data", "run_fusion"};

Environment     = {"POSIXLY_CORRECT=1"};
```

Figure 4.11: JDL file for the plasma visualization application

Neural Network Training

The CrossBroker allows the execution of MPI applications using the nodes of different clusters. Applications that make heavy use of collective operations and are quite sensitive to high latency links are not suitable for this kind of environment. However, there are many applications that exhibit a computation/communication ratio which makes them attractive to an execution distributed over multiple sites. Many embarrassingly parallel applications fit in this kind of application.

ANN is a Master-Worker application developed in the CrossGrid Project to measure the impact of execution using multiple sites. This application performs the iterative process of training a neural network that analyses data from the LEP (Large Electron Positron Collider) at CERN in order to attempt to find the Higgs Boson [104]. The master node assigns a list of files with input data to each of the workers, and the training is repeated until the obtained error reaches a certain bound. The needed files are downloaded from each of the workers using the replica management tools. An example JDL file for executing this application on 10 nodes is shown in Figure 4.12.

The `Executable` attribute defines the binary to be executed, while the `Arguments` attribute contains the name of the file that lists the location of the training files that will be used during execution. This file and the application executables are included in the `InputSandbox` and will be staged on the Worker Nodes by the Job Starter. The `StdOutput` and `StdError` attributes specify the files to which the standard output and error will be redirected. The `OutputSandbox` includes those

```
JobType         = "Parallel";
SubJobType      = "mpich-g2";
NodeNumber      = 10;

Executable      = "ann";
Arguments       = "train_set.xml";

StdOutput       = "std.out";
StdError        = "std.err";
OutputSandbox   = {"std.out","std.err"};
InputSandbox    = {"train_set.xml", "ann"};

Rank            = other.GlueHostBenchmarkSI00;
```

Figure 4.12: JDL file for the ANN application

files, which will be transferred back from the temporary working directory on the execute machine to the submit machine. CEs are sorted according to with the `Rank` expression (`GlueHostBenchmarkSI00` refers to Average SpecInt 2000 benchmark, as an indication of the computational power). It must be noted that the Glue Schema is utilized within the `Rank` attribute, since it is also used for the resource description. Figure 4.13 shows the output of the 'job-list-match' command. It returns a list of resources or groups of resources upon which the parallel job can be executed. Such a list is composed of:

- Groups of elements that contain only one CE, so the job could be submitted to just one CE or cluster. This is the most desirable situation. For example, the CE ce001.grid.ucy.ac.cy:2119 has all 10 free CPUs and a global rank (based on the SI00 of that CE) of 650. This resource will be the first selected by the Scheduling Agent.

- After single CEs, groups of CEs that fulfill the requirements are formed. In this case, we find 2 groups with 2 CEs suitable for executing our job. The best one, according to the rank is the group formed by cagnode45.cs.tcd.ie and ce100.fzk.de. The computed rank of 440 is not the average of the ranks of the components ($(400 + 460)/2 = 430$), but the weighted rank calculated by considering the number of free CPUs of each component.

- As can be seen, there are no possible groups of 3 CEs that contain the required number of CPUs not taken into account in the previous groups (with 1 or 2 CEs). However, we find 1 group with 4 CEs.

```
             GROUPS OF CE IDs LIST

 The following groups of CE(s) matching
 your job requirements have been found:

*Groups with 1 CEs*   *TotalCPUs* *FreeCPUs*

[Rank=650]
 ce001.grid.ucy.ac.cy:2119    10          10
[Rank=630]
 cluster.ui.sav.sk:2119       16          16
[Rank=400]
 zeus24.cyf-kr.edu.pl:2119    58          57

*Groups with 2 CEs*   *TotalCPUs* *FreeCPUs*

[Rank=440 TotalCPUs=12 FreeCPUs=12]
 cagnode45.cs.tcd.ie:2119      4           4
 ce100.fzk.de:2119             8           8
[Rank=498 TotalCPUs=10 FreeCPUs=10]
 ce01.lip.pt:2119              2           2
 ce100.fzk.de:2119             8           8

 *Groups with 4 CEs*  *TotalCPUs* *FreeCPUs*

[Rank=435.6 TotalCPUs=10 FreeCPUs=10]
 cagnode45.cs.tcd.ie:2119      4           4
 ce01.lip.pt:2119              2           2
 cg01.ific.uv.es:2119          2           2
 cgnode00.di.uoa.gr:2119       2           2
```

Figure 4.13: Results of matchmaking

4.5 Conclusions

The CrossBroker is an implementation of a Grid Resource Management System for interactive and parallel jobs following the architecture proposed in the previous Chapter. In this Chapter, we described each of the modules that compose the scheduler:

- The Scheduling Agent is in charge of receiving the users jobs and makes the appropriate scheduling decisions to execute those jobs on the resources. It includes a multi-programming mechanism based on Condor that allows the execution of interactive applications as soon as they are submitted.
- The Resource Searcher discovers the resources available in the grid environment and does the matching between the jobs requirements and the resources. It keeps the resource information in a cache that gets updated asynchronously using a variety of methods.
- The Application Launcher, Job Starters and Interactive Agent are in charge of the actual execution of the application at the resources. The Application Launcher uses Condor-G as a front-end to the Grid. Mpi-Start is used as a Job Starter for parallel applications. We provide a Condor Bypass Interactive Agent, however other agents, such as glogin, have been adapted to be used with our system.

We have also shown examples of real applications that use the CrossBroker features for executing parallel code and interact with the user online.

CHAPTER 5

CrossBroker Experimental Evaluation

In this Chapter, we report on the different experiments performed in order to show the benefits of the CrossBroker mechanisms and the overhead introduced by our middleware on a real testbed.

In Section 5.1 we study the overhead introduced by the CrossBroker and the software stack used in the execution of jobs. This metric is especially important for interactive jobs, where response time should be as short as possible. Section 5.2 presents the usage statistics of the CrossBroker on the int.eu.grid testbed. This study serves as the basis for the creation of the workloads used in the next section, where simulation is used to present the benefits of the proposed mechanisms.

5.1 CrossBroker Overhead

Overhead is a metric that reports delays between the submission and the execution of jobs. This metric is especially important for interactive jobs, where response time should be as short as possible.

In the submission process we have identified the following steps that can cause delays:

1. Job Preprocessing. This is the starting point for every job: the user from the User Interface submits a job described in a JDL file to a CrossBroker.
2. Selection of Resources. In this step, the CrossBroker selects the best resource available to run the job. Some jobs may wait in the scheduler's queue before processing.
3. Remote Job Submission. Once the best resource is selected, the job is submitted to the remote resource using the corresponding Application Launcher.
4. Job Start up and Execution. At the remote resource, the Job Starter prepares the environment for the job, stages all the needed files and starts the job.

In the following Sections we will evaluate the overhead introduced by each of the steps described above.

5.1.1 Job Preprocessing

The submission of jobs is handled by the User Access Module of the Scheduling Agent. The overhead introduced by this step of the job submission process is due to the authentication and authorization process of GSI, the CrossBroker validation of the user request, and the file staging from the User Interface to the CrossBroker. File transfer is done using gridFTP [90], a standard file transfer protocol and server for grids. All the jobs stage at least the user proxy that will be used later for authentication at the remote sites. If the job includes any other files in the `InputSandbox`, those will be also staged. Each file is sent independently, therefore the time increases with the file sizes and the number of files. The CrossBroker includes a configuration parameter that limits the size of the Input Sandbox for every job. By default, this parameter is set to 1MByte. Jobs that may need bigger files should use another transfer mechanism from the Worker Nodes during job execution, such as the Data Management Tools from EGEE [105].

The time spent in this phase is determined by the expression

$$T = T_{auth} + \sum_{i=0}^{n}(lat + \frac{size_i}{bw})$$

where T_{auth} is the time spent authenticating and validating the user request, lat is the network latency, bw is the network bandwith, $size_i$ is the size of file i ($size_0$ is the size of the user proxy that is always transfered), and n is the number of files in the input sandbox.

In our experiments we have a CrossBroker installed at UAB, and we used two User Interfaces: one installed also at UAB, on the same network as the CrossBroker, and

a User Interface located in Germany, on the int.eu.grid FZK testbed. Four types of job descriptions were submitted in sets of 100 jobs to measure overhead:

1. Job without input files.
2. Job with one input file of 64KBytes.
3. Job with one input file of 1MByte.
4. Job with four input files, of 256KBytes each.

Figure 5.1 shows the average time in seconds spent for each kind of submission from the User Interface. The "No files" case illustrates the overhead of the authentication and authorization process, the transmission of the user proxy, and the CrossBroker validation of the request. This is the minimum possible overhead for the job preprocessing step. The local network connection causes an overhead of 2.92 s, and the FZK connection takes 7.89 s. By analysing the obtained results and using the expression proposed above, we have found that T_{auth} is always less than 4 times the network latency.

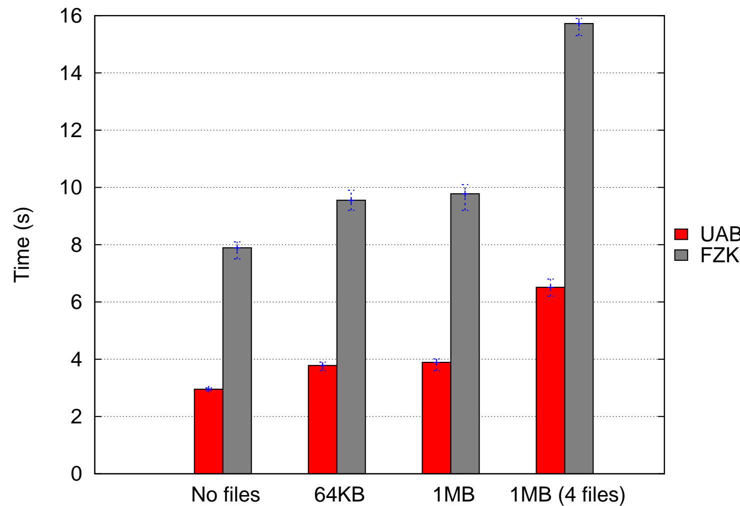


Figure 5.1: Average Overhead for Job Preprocessing

The difference between the "No files" case and the submissions with files are caused by the gridFTP transmission of files between the UI and the CrossBroker. Since each file is transmitted independently and the size of the files is limited to 1MByte, the overhead depends mostly on the latency of gridFTP. Therefore, as the number of files to be transmitted increases, the overhead of user submission will increase. The "64KBytes" and "1MByte" cases have similar overhead, while the case with four files has the largest overhead.

5.1.2 Selection of Resources

The Selection of Resources in the CrossBroker uses the Resource Cache of the Resource Searcher. This allows the fast selection of resources, since there is no need to query the individual resources every time a new job is submitted. Although the use of such a Resource Cache may produce errors in the scheduling decisions due to outdated information, this can also occur without a Resource Cache. The information provided by the resources is not always up to date and depends on the updating mechanisms used in each site. For example in the int.eu.grid project, the information is updated at the sites every four minutes. The Resource Cache alleviates the lack of perfectly up to date information from the sites with the most recent information from jobs submitted by the CrossBroker (selected CPUs are not considered free until the job is returned).

The matchmaking algorithm for intra-cluster jobs has linear complexity; each of the sites in the list is checked once. However, the inter-cluster algorithm must construct the possible sets of machines that may execute the job. The maximum number of sets to be evaluated is $2^n - 1$, with n being the number of resources. This is a worst case scenario since the algorithm discards the solutions in which a subset is already included in a previously matched group and a limit to the maximum number of CEs per set exists.

Our experiment asked for 100 CPUs without any special requirements on the resources. We used two different testbeds: the int.eu.grid (i2g) testbed where 10 sites are available in the Resource Cache, and the EGEE testbed, where 363 resources are available. Figure 5.2 shows the average selection time for submission when the selection used both intra-cluster and inter-cluster allocation on each of the two testbeds.

The matchmaking in int.eu.grid takes less than 0.1 s for intra-cluster selection, while for inter-cluster jobs the elapsed time is 0.52s. The EGEE test, suffers from the need to to check a large number of resources. The time elapsed increases to 0.25 seconds for intra-cluster case and 2.8 seconds for inter-cluster case.

When the Resource Cache is not enabled, there is a query to every site is performed in order to check its attributes and match with the job. This query suffers from timeouts and connection errors and further delays the matchmaking phase. We measured the querying process for the int.eu.grid and EGEE testbeds, discarding all cases with time-outs or errors. In the case of the int.eu.grid testbed, the average time needed for query all the sites is 0.75 seconds (with a standard deviation of 0.01). However for the EGEE testbed, it takes 89.4 seconds on average (with a standard deviation of 2.1) to fetch the information from all the resources. These times should be added to the selection times already shown in the figure.

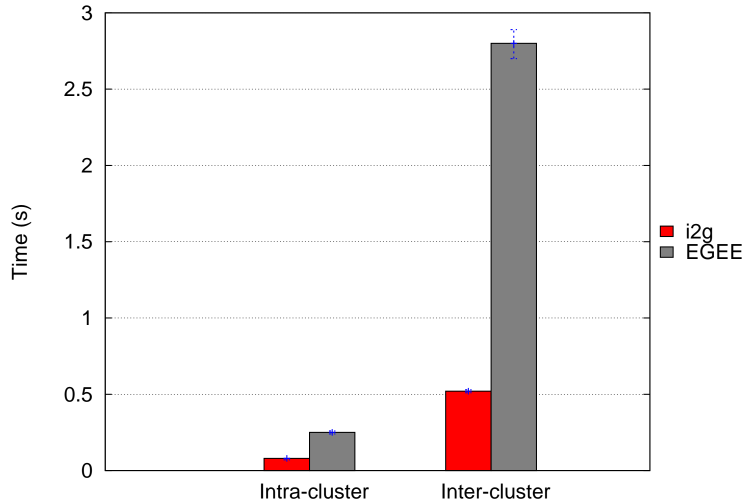


Figure 5.2: Average Selection Overhead

5.1.3 Remote Job Submission

Once resources have been selected by the Scheduling Agent, the Application Launcher invokes Condor-G for submission to the remote resources. In this step, Condor-G contacts the remote Globus GRAM job manager that will in turn submit the job to the local resource management system. This local resource management system will finally select the Worker Nodes and start the application. The overhead introduced by this step depends on the different overheads introduced by the middleware involved. Since the Job Starter is submitted instead of the final user job, the file staging in this step is the same for every job. When using the Glidein mechanism, Globus and the Local Resource Management System are bypassed, starting directly the job in the resource. The only overhead is due to the Condor job starting mechanism.

Figure 5.3 shows the average time in seconds spent during submission to the remote resources using different sites. The CrossBroker machine is installed at UAB, while the remote sites are at UAB, FZK (Karlsruhe, Germany), LIP (Lisbon, Portugal), and BIFI (Zaragoza, Spain). Submission occurred when the Local Resource Management System queues were empty, hence the jobs did not wait in the queue. The submission time for jobs using Glidein for the same sites is also shown.

The overhead in remote job submission using Globus is given by the following expression:

$$T = T_{globus} + T_{jobmanager} + T_{queue}$$

where T_{globus} is the time spent in contacting the remote resource and cimpleting the

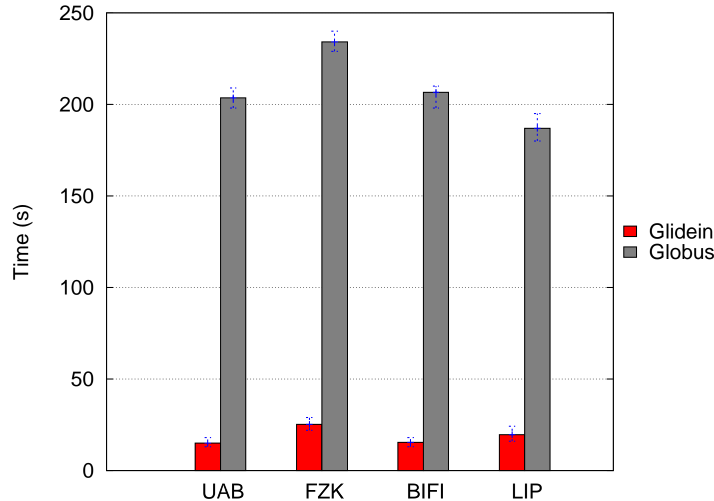


Figure 5.3: Average Remote Job Submission Overhead.

request for a new job submission. This time is dependent on the network connection between the sites and the CrossBroker. We have found in our experiments that T_{globus} reaches the maximum of 28 times the network latency. $T_{jobmanager}$ is the time spent by the jobmanager preparing and submitting the job to the Local Resource Management System (LRMS). UAB, FZK, and BIFI use Torque as their LRMS, while LIP uses SGE. The jobmanager for SGE is developed at LIP and is optimized for the site, explaining the lower overhead. In our test, this time is an average 180 s for the Torque jobmanager, and 150 s for SGE. The last term of the expression, T_{queue}, is the time spent in the queue before execution. In our experiments this time is negligible since the queues are empty at the time of submission.

In the case of the submission to Glidein, there is no jobmanager that deals with the creation of the job and there is no queue for starting the jobs. Therefore, the overhead is given by the time spent by Condor to start the job at the remote site. Again, this time depends on the network connection, hence FZK and LIP have a higher overhead due to slower network connections. Experimentally, the time needed for creating the job at the remote site is less than 15 times the network latency. Submission via the Glidein is less than 30 seconds for the sites, which is fast enough to provide a good interactive experience.

5.1.4 Job Start up and Execution

Job Start up is the time elapsed from the start of the Job Starter on the remote Worker Nodes to the user application start. Two cases can be considered here:

batch jobs that set up the Glidein mechanism and run in one of the virtual slots, and jobs that run directly on the resources assigned.

The creation of the multiprogramming environment includes the download of the needed Condor binaries, the configuration of the Condor environment to execute the jobs, and finally the start of the virtual slots. Figure 5.4 shows the time elapsed from the allocation of the machine by the Local Resource Management System to the execution of the batch job on the virtual slot. As before, the times for UAB, FZK, LIP and BIFI sites are depicted. Two files are downloaded to the Worker Nodes: a bundle of condor binaries in a single compressed tar archive (5.4 MBytes) and configuration file that is customized at the Worker Node (5 KBytes). Once the files are downloaded, the configuration time is negligible and not shown in the figure. The major overhead contribution is the time needed for Condor to start and contact the CrossBroker and the final execution of the job. This time is less than 44 times the network latency. The total overhead is under 90 seconds even for the farthest site, which is acceptable for batch jobs that are expected to run for hours.

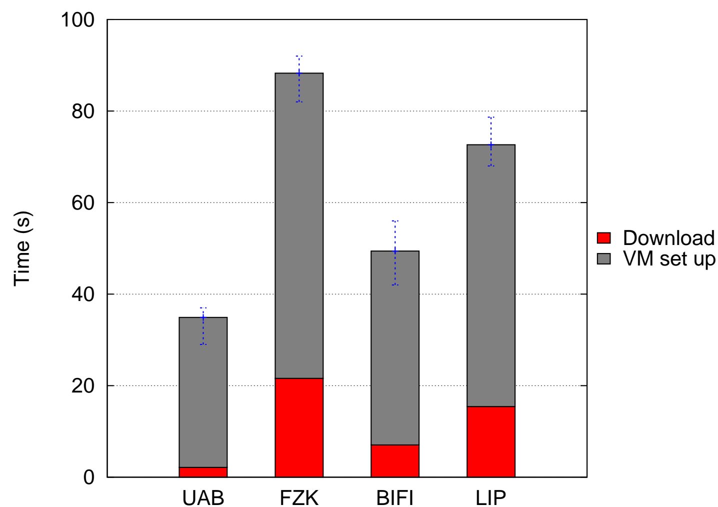


Figure 5.4: Average Glidein Start up Overhead

Once the job has a machine allocated for execution, the Job Starter prepares the environment for running the application. The major contribution to the overhead in this step is the download of input files. As discussed previously in Section 5.1.1, the download times depend in the size and number of files transferred and the network connection between the CrossBroker and the Worker Node.

Applications running on a virtual slot created by Glidein have an extra overhead due to the Condor daemons running in the Worker Nodes and the job running in the other

virtual slot. The overhead due to the execution of the application on the virtual slot depends greatly on the type of application and its priority adjustment. We measured the impact of the multiprogramming environment using one of the applications of the int.eu.grid project. This application, called eIMRT (enhanced Intensity Modulated Radiation Therapy,) performs numerical calculations in order to determine the best set of radiation beam trajectories, their intensity, and their shape configurations in order to offer the best treatment to the patient. It is a CPU-intensive application that simulates the various possibilities for the radiation beams. It has an initial I/O phase to read large input data files with the patient information. The application was executed with a WN under the following scenarios:

- Bare machine, without any Glidein mechanism running.
- Application running in the batch virtual slot, without any application running on the interactive virtual slot and vice versa.
- Running on one virtual slot, while sharing the machine with another application on the other virtual slot. A study of the priorities of both slots was performed in this case.

Figure 5.5 compares the average execution time for these scenarios, were the application is running without sharing the machine. Each scenarios was was executed 100 times. In our experiments we have that the Condor overhead is negligible: applications running with or without the Condor daemons have no differences in run time if there is no other application running on the other virtual slot.

In the last scenario, the type of application executed (I/O or CPU-intensive applications) concurrently determines the overhead. We used a synthetic CPU-intensive application with a 98% CPU executed together with the eIMRT. Figure 5.6 depicts the influence of this synthetic application on the run time of the eIMRT application. The average run time in seconds of the eIMRT is plotted against its priority. The priority ranges from 19, the minimum, to 0, the maximum. These are user settable scheduling priorities in the operating system. Five different cases are shown, identified using the priority of the synthetic application (0, 5, 10, 15, and 19). The black dashed line depicts the average run time of the application when running without sharing the CPU. The run time for eIMRT priority 19 with synthetic application priority 0 has not been plotted for figure clarity: in this case the run time is 12582.14 seconds.

As can be seen in the figure, once priority reaches a certain level, the gains in run time are not significant: assigning a higher priority to the job does not greatly affect the run time. This priority level depends on the applications sharing the physical machine. In our current configuration, the batch virtual slot is assigned a priority of 10, while the interactive job is 0. This is the way we ensure a low overhead

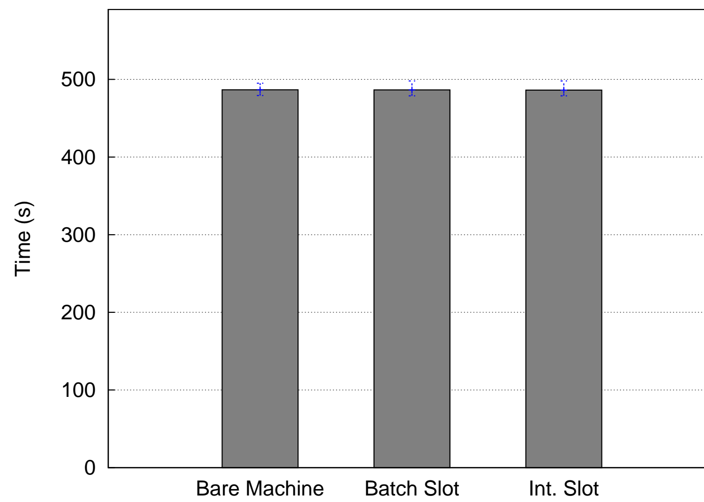


Figure 5.5: Average Execution Time of eIMRT in the Glidein environment without sharing the CPU

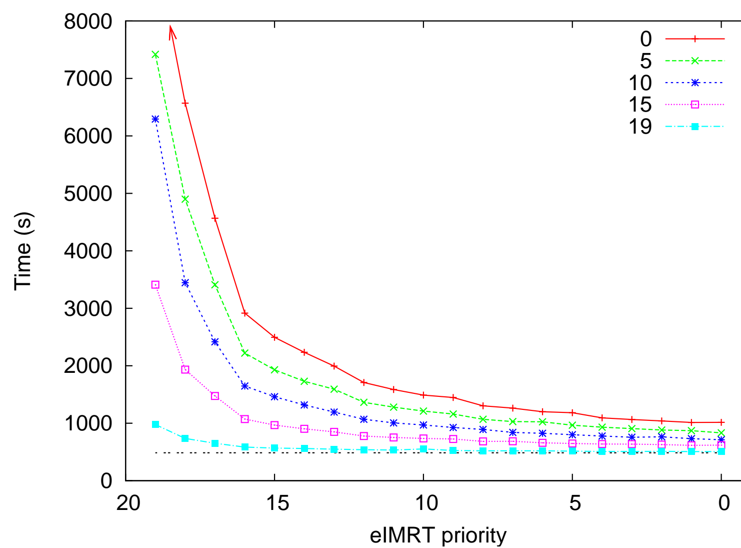


Figure 5.6: Runtime of eIMRT on virtual slots with different priority

for interactive priority: in our experiments with the synthetic application priority 10 and eIMRT with priority 0, the increase in run time is less than 30%. If the eIMRT application executes with priority 10 and shares the CPU with a high priority application (red line in the plot), the run time is increased by 210%. Note that we are considering a worst case scenario, since applications (specially interactive) are usually not that demanding of the CPU. Moreover, interactive applications are usually short, so the batch application usually does not share the CPU during its entire execution.

5.1.5 Overall Overhead

As seen in previous sections, the overhead of the applications submitted to the CrossBroker greatly depends on the overhead introduced by the different middleware components used at each step: gridFTP for the transmission of files, Condor and Globus GRAM for the submission to the sites, and the LRMS for the final resource allocation. The CrossBroker introduces mechanisms to completely avoid the overhead produced by GRAM and LRMS for interactive jobs using Glidein mechanisms.

An interactive application without input files (e.g. a shell session on a grid resource) submitted from a UI with a slow network connection to the CrossBroker, on a testbed where there are more than 300 different sites that gets submitted to a site with a slow network connection such as FZK, the application can start in less than 40 s if there is a virtual slot available. In the case of needing to pass through Globus and the LRMS at the remote site, the submission time will increase to 240 s. Using input files increases the time needed for starting the applications.

Batch applications have larger overheads: they always go through Globus and they may start the Glidein mechanism. However, the impact is low for applications that run for a long time. In a worst case scenario, for an application submitted to a site selected from more than 300 sites with 1MByte of input files, the total waiting time would be around 380 s if the site queues are empty.

5.2 The CrossBroker on a real testbed

The CrossBroker has been used in the CrossGrid and int.eu.grid projects as the main Grid Resource Manager for submission of parallel and interactive application on the respective projects' testbeds.

All the software, including the CrossBroker, used in both projects had to pass a Test & Validation (T&V) phase where an independent project team would test and assure that all the features described by the software developers were working properly, and that it interacted nicely with the rest of the software. The CrossBroker passed several

Prod Site	Country	Cores	Arch	Storage	LRMS
BIFI	Spain	22	Xeon	0.1 TB	Torque
CESGA	Spain	20	Xeon	4.8 TB	Torque
CYFRONET	Poland	20	Xeon	1.0 TB	Torque
ICM	Poland	32	Opteron	1.3 TB	Torque
IFCA	Spain	314	Xeon	1.1 TB	Torque
IISAS	Slovakia	32	Core Duo	0.5 TB	Torque
LIP	Portugal	57	Opteron	1.6 TB	SGE
PSNC	Poland	107	Itanium/Xeon	4.4 TB	Torque
FZK	Germany	100	Opteron	1.8 TB	Torque

Table 5.1: Production sites properties

Dev Site	Country	Cores	Arch	Storage	LRMS
FZK	Germany	4	Xeon	3.4 GB	Torque
TCD	Ireland	26	PIII	3 TB	Torque
UAB	Spain	20	PIV	380 GB	Torque
GUP	Austria	10	Xeon/AMD	500 GB	Torque

Table 5.2: Development sites properties

of these T&V tests, the most recent one in January of 2008.

5.2.1 The int.eu.grid testbed

The int.eu.grid infrastructure is split into two independent grid computing services called *production* and *development*. The production service is composed of nine clusters and provides a reliable environment to run end-users' applications. Each cluster deploys different CPU architectures (Xeon, Opteron, Pentium D, and Itanium), and uses their preferred Local Resource Management System (LRMS) for job management control.

The development infrastructure is intended to provide a realistic, yet flexible environment for development activities. It is composed of four clusters, and is used to support the testing and integration of the middleware. For higher flexibility and to maximize the available resources, development sites makes heavy use of virtual machines.

Tables 5.1 and 5.2 provide a brief summary of the sites' most important features, namely their capacity, cluster architecture, and LRMS.

The production infrastructure has a total of 700 CPU cores and 16 Tbytes of

storage, while the development infrastructure has 60 CPUs and less than 4 Tbytes of storage. For both infrastructures there is a set of core services that provide resource management (the CrossBroker), information repositories for resources and files (one MDS top-BDII and an LFC replica catalog), monitoring services (an RGMA server), and authentication services (VOMS and MyProxy servers). Core services for the production testbed are located at LIP (Portugal) and replicated at IFCA (Spain). In the case of the development testbed, they are located at FZK (Germany).

5.2.2 CrossBroker Usage

There is no global accounting for all the jobs submitted to the testbed during the lifetime of the project: most of the accounting tools were developed during the project, hence only the last months of int.eu.grid provide data about the CrossBroker usage.

In Figure 5.7 the number of jobs submitted to the production testbed from March 2007 to June 2008 is depicted. The jobs are classified according their type: normal, parallel intra-cluster jobs using Open MPI or MPICH, and parallel inter-cluster jobs using PACX-MPI. The periods of maximum activity match with the project reviews and meetings (November 2007 and March 2008). A total of 442513 jobs were submitted during this period. Normal jobs are predominant (a 65%), especially at the beginning, where users were not yet familiar with the parallel job features and the testing was done primarily on the development testbed. Intra-cluster jobs represent 24% of the total number of jobs, while inter-cluster jobs represent 11%. Although not shown in the Figure, interactive jobs account for 5% of the total number of jobs submitted.

Figure 5.8 shows the percentage of parallel jobs for different job sizes. Jobs with 2, 3, and 4 CPUs account for 98% of the submitted parallel jobs. For larger job sizes, the most frequent ones are jobs with 8 and 10 at 0.36% and 0.38% respectively of the total number of parallel jobs submitted.

Table 5.3 reports the relevant job states as seen from the production CrossBrokers for the same dates, as well as the number of jobs going through each of those states. During the time period considered in this analysis, 442513 jobs have been successfully registered, meaning that they were successfully authenticated and authorized. Only 1.5% were not accepted by the CrossBroker due to bad use of submission protocols, syntax errors, or to problems with the input sandbox file transfers. After job acceptance, the requirements for each job are matched against the available infrastructure resources, resulting in a list of possible CEs on which to run the job. The highest number of job submission failures occurs during this step (28.3%). Analyzing these failures, it can be concluded that most of them are caused by incoherent user specified requirements, such as requesting features that do not exist, or asking for

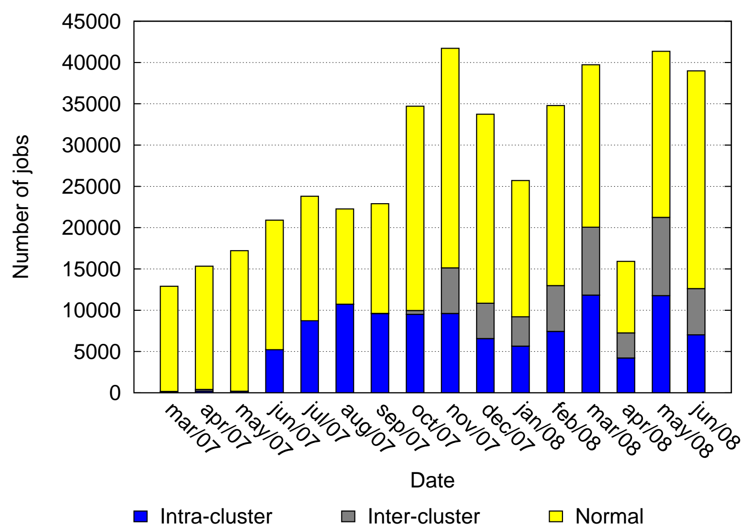


Figure 5.7: Jobs submitted to the int.eu.grid infrastructure.

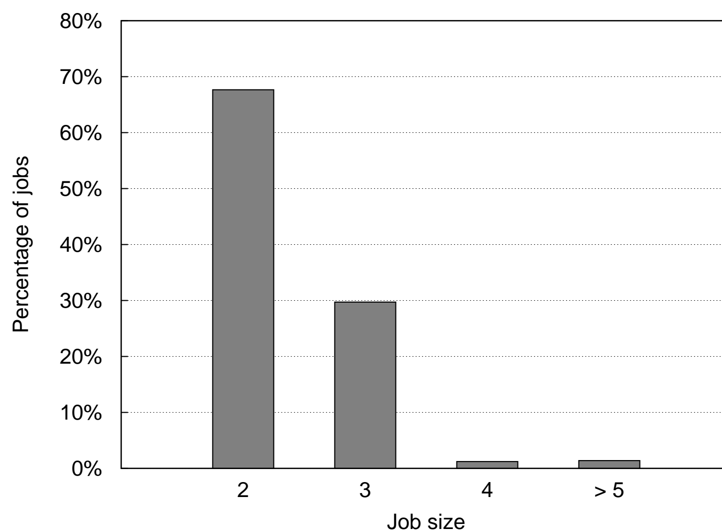


Figure 5.8: Jobs sizes in the int.eu.grid infrastructure.

resource quantities not available at a given time. Another source of failure are the interactive requirements. Interactive jobs that do not receive resources for executing immediately (with or without Glidein) fail to pass to the next scheduling step.

Job States	Total Jobs	Fraction (%)
Submitted	442513	100.0
Accepted	435875	98.5
Matched	310644	70.2
Run	303564	68.6
Completed	296484	67
Cleared	261968	59.2

Table 5.3: Number of jobs processed by CrossBroker

From the total number of jobs submitted to the remote sites (310644 jobs), 97.7% start execution properly and 95.4% end successfully. On the one hand, failures to start are due to problems at the site level. A wrongly configured site can reject jobs or start them incorrectly (i.e. not using the requested number of CPUs). On the other hand, for a job to end successfully, the user application must end correctly and there may be no errors in file staging at the end of execution. Finally, a *Cleared* job status stands for jobs where the output sandbox files have been retrieved by the user, an action which is clearly dependent on the user side.

The total CPU consumption of all the jobs submitted during that period is around 56% of the total CPU power. In Section 5.3, an evaluation of the CrossBroker scheduling mechanisms and strategies is performed using simulations.

5.3 Evaluation of the CrossBroker Mechanisms

In this section we evaluate the proposed CrossBroker mechanisms using simulation. First, we study the modelling of a realistic workload for a grid environment such as the int.eu.grid testbed. Then Section 5.3.2 reviews some of the simulators available for grid scheduling, followed by giving the results of the simulation of the execution of parallel and interactive jobs with the CrossBroker.

5.3.1 Workload Modelling

One of the most important factors that contribute to the performance of large-scale systems are the queueing effects due to the random nature of the demand. This is dependent on the nature of the system's workload. The Parallel Workload Archive

(PWA)[106] collects parallel supercomputer traces from real systems. It is the current de-facto standard source of workload traces and it has been used extensively in research on parallel schedulers and grids. However, there exist significant differences between the parallel supercomputer workloads and the typical workloads of real grid environments.

The Grid Workload Archive (GWA) [107] is an ongoing effort for collecting realistic workloads that can be used in research. Created in 2006, the GWA currently contains traces of nine well-known grid environments, with a total content of more than 2000 users submitting more than 7 million jobs over a period of 13 operational years, and with working environments spanning more than 130 sites comprising 10000 resources.

There are two important characteristics of the traces collected by the GWA:

1. The percentage of single processor jobs is much higher in GWA than in the PWA. There exist 70% – 100% single processor jobs in GWA. Even an infrastructure such as int.eu.grid, which is designed for execution of parallel jobs has around 65% of single processor jobs, as shown in Section 5.2.2.

2. The grid single processor jobs typically represent instances of batch submission. A batch submission is a set of jobs ordered by the time at which they arrive at the system, where each of the jobs is submitted at most Δ seconds after the first job ($\Delta = 120s$ is considered the most significant). The run time of jobs belonging to the same batch submission is large, however the user submits these jobs with a single run time estimate, severely affecting scheduling policies that rely on user estimates (backfilling).

The GWA traces cannot be used unmodified on systems different from the one originating the trace, since the size of the systems is different and the job submission depends on the original circumstance. Moreover, modifying the real traces by scaling or duplicating their jobs may lead to input that does not actually represent a realistic trace, thus affecting scheduling results [108].

In order to address these issues, we employ a model-based trace generation for our experiments. We use the Lublin model [109], which is extensively used by the resource management research community. This is a model for rigid jobs, i.e. jobs whose size is fixed upon arrival to the system. The model includes an arrival pattern with a daily cycle, and a distinction between interactive and batch jobs. The run times of jobs in the model are correlated with the number of nodes of each job. The job parallelism is modelled on two classes: single-processor jobs, and parallel jobs. In our case, the probability of single-processor jobs, p, is fixed to reflect the values found in int.eu.grid, $p = 0.65$. The remainder of the jobs (the parallel ones) have a maximum of 128 nodes, and the average is 4 nodes.

5.3.2 Simulation of grid environments

In a grid environment, it is hard or even impossible to perform scheduler performance evaluation in a *repeatable* and *controlled* manner due to their large dimensions and dynamic nature. To overcome this limitation, a simulation tool can be useful in evaluating the behavior of the CrossBroker. Simulation has been used extensively for modelling and evaluation in real world systems. Several projects have developed tools to study the design and operation of grid environments. The most notable ones are: Bricks [110], Simgrid [111], and GridSim [112].

- Bricks, developed at the Tokyo Institute of Technology, is designed to investigate scheduling issues. It provides a centralized scheduling methodology and allows the use of external tools for network modelling. The computational resources of the given global computing system are parameterized by performance, load, and their variance over time. This restricted model of the resources does not make it suitable for simulating our system.
- SimGrid was developed initially at the University of California at San Diego. It is an event driven simulator written in C. It supports the modelling of resources that are time-shared using a reference resource. The performance of any of the resources is specified with respect to this reference resource, used as a ratio to scale submitted jobs. It has been used in several studies to evaluate scheduling policies, but due to its design, it is difficult to simulate resources with their own policies without extending the toolkit substantially.
- GridSim is a Java simulator based in the SimJava library. It is designed for simulating heterogeneous resources with several users, applications, and local schedulers. It provides network topology simulation [113], the use of workload traces, and the detailed simulation of parallel applications [114]. The source code is freely available and has a detailed documentation.

We used GridSim for simulating our system. We introduced changes necessary in order to model the multiprogramming environment created by the Glideins on the remote resources and implemented a CrossBroker scheduler that performs the same actions as the real implementation. Using the GridSim tools, we simulated the production int.eu.grid testbed with the characteristics shown in Table 5.1: 9 sites with a total of 686 CPUs.

In the simulation we considered the following parameters:

1. **FCFS scheduling policy at the cluster level.** All of the sites in the int.eu.grid testbed use FCFS in their LRMS. Moreover, policies that rely on runtime estimates are difficult to implement on grids.

2. **No background load.** We assume that all load on the system originates from the submission of jobs to the CrossBroker. In the case of the int.eu.grid project, all the load is generated by the CrossBroker.

3. **Simplified network.** We simplified the interconnection network that is available in the int.eu.grid testbed and considered a single router per country, connected to all the sites in that country. All those routers connect point to point.

5.3.3 Co-allocation and Parallel Jobs

Parallel jobs executed in grid resources may use many processors and take advantage of using different resources available in a grid. The co-allocation mechanisms of the CrossBroker allow the execution of such applications automatically, harnessing the infrastructure potential in an efficient way. In this section we evaluate the benefits of using the CrossBroker for executing parallel jobs on grids.

In order to evaluate the co-allocation features, we used the GridSim simulator with extensions for simulation of parallel applications. These extensions allow the definition of jobs composed of tasks. Each one of the tasks is assigned to a CPU and can communicate with the others, taking into account the underlying network. For our simulation we have specified two kinds of parallel jobs:

1. CPU-bound jobs modelled as Master/Worker applications, where the Master distributes a set of tasks to the Workers and gathers all the results once the computation has finished. The size of the tasks depends on the run time of the job as defined by the workload model, and the communication depends on the number of nodes used by the application.

2. Communication-bound jobs modelled as SPMD applications. In this case, communications occurs between adjacent neighbouring tasks every computing iteration. The number of iterations depends on the length of the job, while the communication is fixed for each iteration (4096 Bytes).

A workload of one day is generated with the Lublin model and without interactive jobs. Figure 5.9 shows the job submitted per hour in intervals of 15 minutes for the generated workload. The Lublin model creates cyclic arrival patterns that repeat for each day. The results are similar for workloads that elapse over longer periods of time. This workload generates a high load during the day, with the submission of more than 70 jobs in one hour. CPU-bound and communication-bound jobs are distributed with equal probability within the workload.

We have simulated the same workload considering three different scenarios. Scenario A simulates a situation without co-allocation. Therefore, jobs must wait until

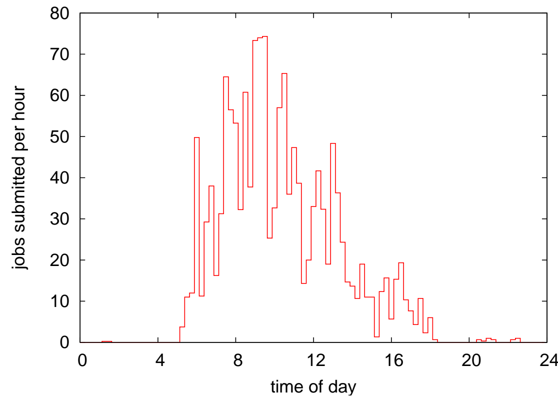


Figure 5.9: Job arrival pattern in the workload.

there are enough free CPUs within one resource. Scenarios B and C simulate co-allocation. Co-allocation allows jobs to be executed if there are enough free CPUs on one resource, or if there is a set of resources with sufficient CPUs to execute the job. Scenario B co-allocates jobs without considering the characteristics of CPU-bound or communication-bound. Scenario C co-allocates only CPU-bound jobs.

In order to evaluate the influence of the selection of nodes for execution of parallel jobs, three different selection policies were implemented in the CrossBroker simulation:

Worst Fit (WF). The Worst Fit allocation policy selects the site with the largest number of free CPUs to execute a job.

Best Fit (BF). In this case, the site with smallest difference between the CPUs requested and the number of CPUs available at the resource is selected.

First Fit (FF). The First Fit policy selects the first site considered with sufficient CPUs to execute the job.

Inter-cluster execution is only used when there are insufficient resources at a single site for the execution of the job. The CrossBroker always tries to minimize the effect of the high latency network links between sites for inter-cluster jobs by selecting the set with smaller number of resources. In order to select the best set from the ones with smaller cardinality, a Best Fit policy is used, creating the largest job components possible.

Figure 5.10 shows the system utilization during the execution of the workload for the **Worst Fit** policy. The value 1 implies that all the CPUs in the system are being used for execution of jobs, and 0 implies that all the CPUs are free. Scenario A is

shown in red, scenario B in green and Scenario C in blue. The bottom figure shows the evolution of the average system utilization during the simulation. In the top figure it can be seen that *scenario A* (without co-allocation) is not able to efficiently use the resources. When a large job is to be executed and there are not enough resources, the rest of jobs are held in the queue. Once enough CPUs have been freed for the first job, the following jobs also enter the system until another large job holds the queue, producing the peaks of system load seen in the figure. In *scenario B*, where the CrossBroker uses co-allocation whenever there are not enough machines on one single site to execute the first job in the queue, the utilization of the resources reaches 100% most of the time. *Scenario C* sees an intermediate result, with some Communication-bound jobs holding the queue.

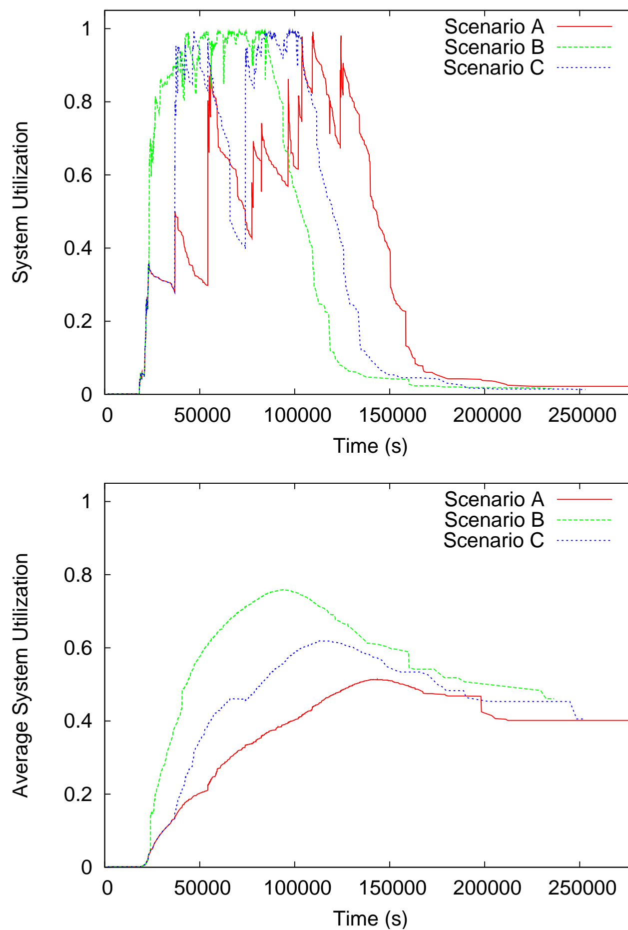


Figure 5.10: Top: Worst Fit allocation policy system utilization. Bottom: Average system utilization for Worst Fit allocation policy

Figure 5.11 shows the system utilization versus time for the **Best Fit** policy of the three simulated scenarios. Although the Best Fit policy produces more fragmentation at smaller sites (i.e. the free CPUs usually are not sufficient for the execution of larger jobs), it tends to use the smaller sites, leaving space for the larger jobs on the bigger sites. Therefore, the big jobs do not hold the queue as frequently as the Worst Fit policy. Regardless of the better system utilization for all scenarios, co-allocation scenarios outperform non-coallocation by making use of the spare CPUs available at the sites where there are not enough CPUs for running the jobs in the queue.

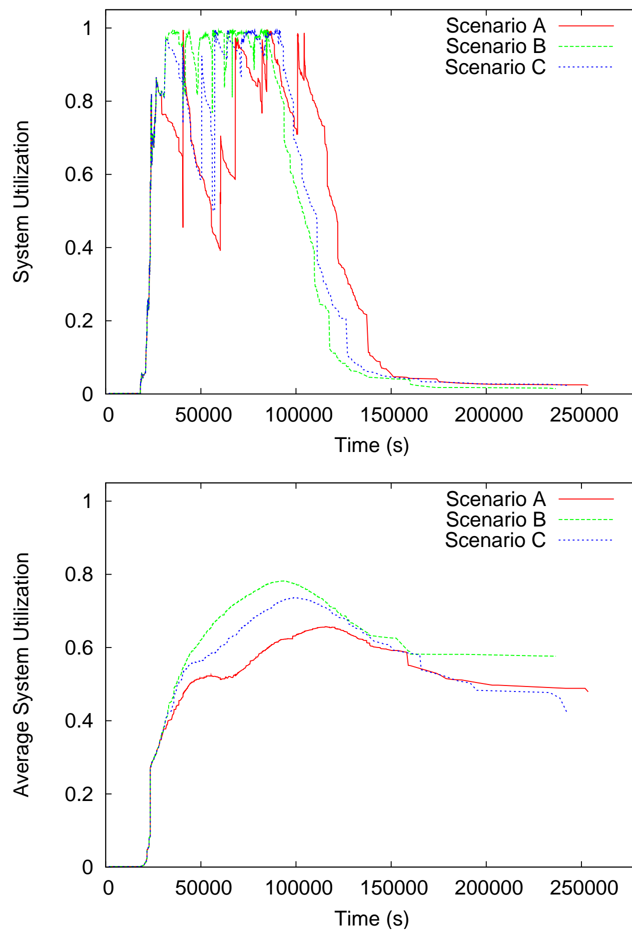


Figure 5.11: Top: Best Fit allocation policy system utilization. Bottom: Average system utilization for Best Fit allocation policy

The behavior of the **First Fit** allocation policy is shown in Figure 5.12 — the top plot shows the system utilization and the bottom plot shows the average system utilization. This policy does not consider the global state of the resources, and only

considers the first resource available for running a job. Therefore it schedules jobs faster than the other policies. Although it schedules jobs without taking into account the status of all resources, it produces better results than the Worst Fit policy and gets better system utilization in all scenarios.

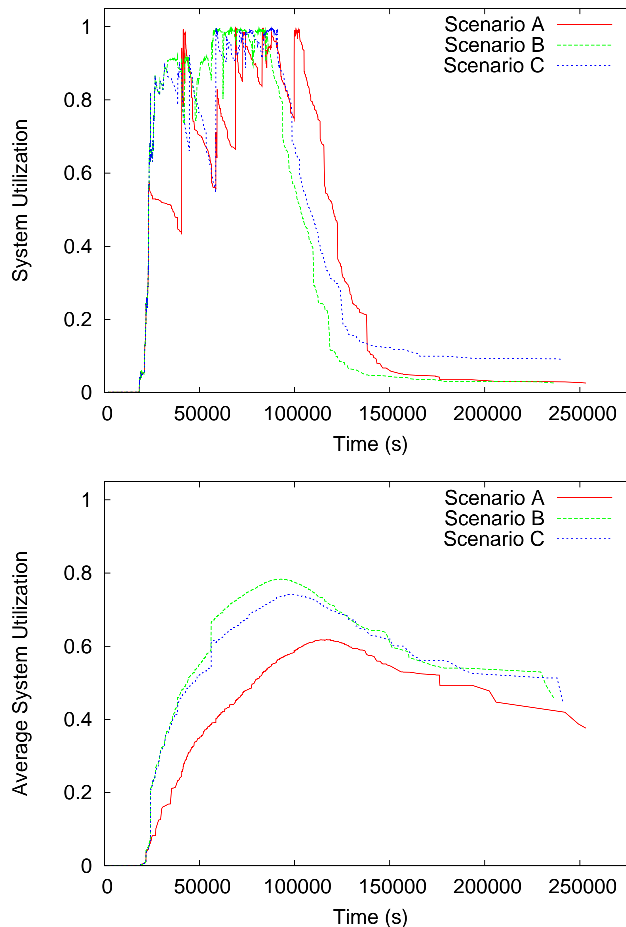


Figure 5.12: Top: First Fit allocation policy system utilization. Bottom: Average system utilization for First Fit allocation policy

Co-allocation improves system utilization, especially when the job type is not taken into account. However, the jobs that are executed using several resources may suffer the impact of low latency links during run time, especially communication-bound jobs. In order to measure the impact of the network on a co-allocated job, we repeated the simulation with homogeneous resources and measured the difference in run time for every job with and without co-allocation using the Worst Fit policy (where co-allocation is used more frequently). Figure 5.13 shows the probability

density function of differences on the run time for the co-allocated jobs. The x-axis shows the percentage of jobs suffering the impact, and the y-axis shows the impact as a function of the expression $\frac{r_{nc}-r_c}{r_{nc}}$, where r_{nc} is the run time of the job when co-allocation is not used, and r_c is the run time of the job when co-allocation is used. CPU-bound jobs are shown in red, and communication-bound jobs are shown in green.

Figure 5.13: Impact on the run time of co-allocated jobs

For a great percentage of CPU-bound jobs, the inter-cluster execution has a negligible impact on their run time (less than a 0.01% difference in run time). Moreover, most (more than 85% of the jobs) increase their execution time by less than 10%. The remainder of jobs suffer a higher impact due to having very short execution times. Note that the CPU-bound jobs are ideal for co-allocation. The impact is greater on communication-bound jobs, especially for short jobs, where communication tasks account for most of the run time. As seen in Figure 5.13, most jobs have between 20% and 80% longer execution times, although very small jobs can reach as high as 700%. Although *scenario C* does not get the best utilization of the resources, only CPU-bound jobs are co-allocated, and therefore all the jobs have similar run times to *scenario A*, where no co-allocation is used. We recommend that communication-bound jobs not to be submitted as inter-cluster jobs.

As seen in Section 4.1.2, the CrossBroker implements a Best Fit policy in its scheduler, which gets good results for the three scenarios in the simulation.

5.3.4 Glidein and Interactive Jobs

The Glidein mechanism enables the fast start-up of interactive applications, even under conditions of high occupancy of the resources. In this section, we present the results of the simulation of this mechanism, and its benefits for interactive jobs. We simulated two different scenarios; the first one with a low load workload, in order to measure the overhead imposed on batch jobs, and the second one with a high load workload in order to show the benefits for interactive jobs.

The theoretical load level of a system is given by the following expression:

$$load = \frac{\sum(r_i \cdot n_i)}{P \cdot \max(a_i)}$$

where r_i is the run time of job i, n_i is number of nodes in job i, a_i is the arrival time (measured from the beginning of the simulation) of job i, and P the number of nodes in the system.

By modifying the parameters of the Lublin model that characterize the inter-arrival time between consecutive jobs during peak hours, we are able to generate specific workloads, for a system of known size. We have 686 CPUs. Using this approach, we generate synthetic job streams, each lasting one week, for high ($load = 0.9$) and low ($load = 0.2$) load. Figure 5.14 shows the job submitted per hour in intervals of 15 minutes for each of these workloads. The arrival pattern is cyclic, thus only a period of 24 hours is shown in the figure. Interactive jobs comprise around 5% of the total jobs in the workload, in concordance with the real workload of the int.eu.grid project (see Section 5.2.2).

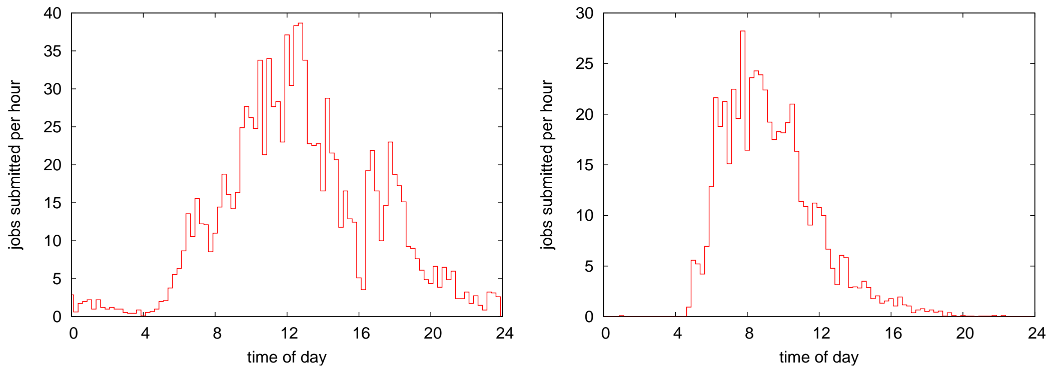


Figure 5.14: Cyclic job arrival pattern in the workloads. Left: High Load. Right: Low Load.

The actual run time of jobs is modeled by a hyper-Gamma distribution with two stages. For parallel jobs, the parameter that represents the probability of selecting

the first hyper-Gamma stage over the second depends linearly on the number of nodes. With these parameters, the average job run time is approximately 30 minutes for interactive jobs and four hours for batch jobs. Figure 5.15 shows the probability density function for the $\log_{10}$ of the run times (in seconds) for batch and interactive jobs.

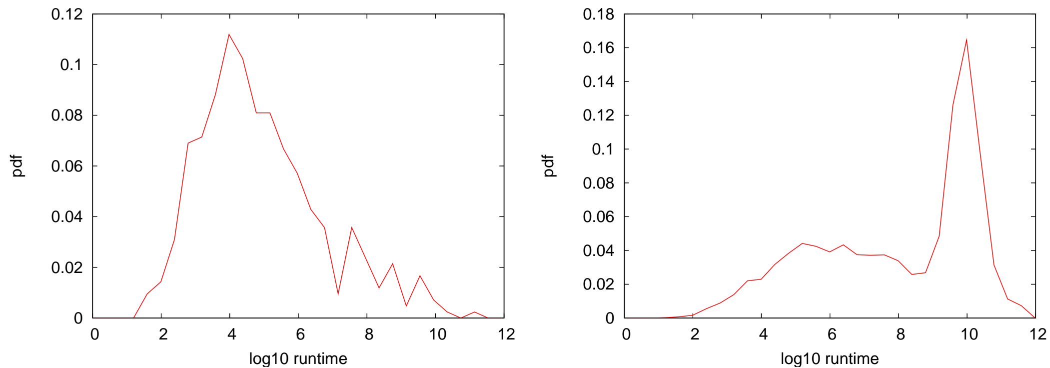


Figure 5.15: Distribution of runtimes in the workloads. Left:Interactive jobs. Right: Batch jobs

In order to evaluate the benefits of Glidein and CrossBroker policies for interactive jobs we simulated the following policies:

FCFS. The CrossBroker employs a First Come First Serve policy for batch jobs, while interactive jobs have better priority and are submitted to sites with enough free CPUs to execute the job.

Glidein + FCFS. As in the previous case, batch jobs are scheduled using a First Come First Serve policy. However interactive jobs use free CPUs or virtual slots already available at the sites. Only serial batch jobs create new virtual slots, therefore the creation of virtual slots overhead only applies to them.

Glidein + BF. Although Backfilling is not feasible due to the inaccuracy of user estimates, we simulated a policy where if some nodes are empty and the first job in the queue is not suitable, the queue is examined for other jobs. Since the scheduler does not have job execution times or user estimates, we restrict the number of allocated CPUs for subsequent jobs to the number of CPUs that this first job needs. For example, if the first job in the queue is waiting for 32 CPUs, and the queue contains three jobs needing 16, 10, and 20 CPUs respectively; the jobs needing 16 and 10 can be executed if there are free CPUs available, since 26 is smaller than 32.

Results for low-load Workload

The low-load workload let us know the overhead of Glidein and how it affects running jobs. Since the workload is low enough to have free CPUs whenever an interactive job arrives at the system, the Glideins will not be used for interactive jobs. However, the serial batch jobs create virtual slots, allowing us to measure the overhead introduced by the system. Table 5.4 shows the most significant metrics for each of the policies:

- Makespan: total execution time (in seconds) of the entire workload. It corresponds to the time when the last job of the workload ends.
- Average SlowDown (ASD): the slowdown is defined as the response time divided by the job size. It gives a measure of system performance, by comparing the execution time of one job with its size. We would like small jobs to have small response times and large jobs to have large response times.
- Average Wait Time (AWT) of batch jobs (in seconds): the wait time is defined as the time elapsed from the job submission until the start of its execution. It measures how long the job has been waiting in the queue. Only the wait time of serial batch jobs is considered here, since they activate the Glidein mechanism.
- Failed: percentage of failed jobs. Interactive jobs without resources at the time of submission are cancelled and considered failed.
- Average Load of the System: The average load measures the utilization of the system. A load of 1 means all the CPUs in all the resources are being used; 0 means no resources are being used.

Policy	Makespan (s)	ASD	AWT (s) batch	Failed	Avg. Load
FCFS	624112	1.39	248.53	0%	16.1%
Glidein + FCFS	660722	1.81	342.48	0%	15.9%
Glidein + BF	652246	1.76	327.54	0%	15.9%

Table 5.4: Results for low-load Workload

The difference in the Makespan between *FCFS* and the other two policies is mostly due to the overhead introduced by the creation of Glideins. This difference is around 5%. This overhead is also reflected in the Slowdown (defined as the response time divided by the run time) and Wait Time. The Average Wait Time of the *FCFS* policy is in accordance with the time needed for submission of jobs when going via Globus and the LRMS in the sites (see Section 5.1.) The difference in wait times is approximately 90 seconds with the other policies, which is the time needed to start the virtual slot mechanisms on the resources. The results for the *Glidein + BF* policy are slightly better than the results for *Glidein + FCFS*, because more jobs

can enter the system. The load level for each of the policies behaves similarly. There are no significant changes in the actual execution time, all the jobs are exclusively executed without sharing the machine with other jobs. No jobs are cancelled because there are sufficient free resources to run all the jobs.

We conclude from these results that the Glidein mechanism does not introduce a significant overhead in the system.

Results for high-load Workload

When the resources are saturated (i.e. all CPUs are being used), it is more difficult to find free CPUs for interactive jobs. Therefore, interactive jobs have a greater chance of being cancelled. Table 5.5 shows the same metrics as Table 5.4 for each of the policies: makespan (in seconds), average slowdown, average wait time for serial batch jobs (in seconds), percentage of interactive jobs failed, and the average load of the system.

Policy	Makespan (s)	ASD	AWT (s) batch	Failed	Avg. Load
FCFS	881509	17.78	3185.32	14.6%	40.5%
Glidein + FCFS	911828	18.75	3376.30	0%	43.7%
Glidein + BF	899773	18.74	3365.49	0%	44.1%

Table 5.5: Results for high-load Workload

The *FCFS* policy without the multiprogramming mechanism fails to serve all the interactive jobs in the workload. 14.6% of them are cancelled due to a lack of free resources available at the time. Makespan is lower in this case, because there are fewer jobs executed and jobs are executed in an exclusive manner. The policies with Glidein achieve a larger makespan (3.4% bigger in the *Glidein + FCFS* policy and 2.1% in the *Glidein + BF* policy), but are able to handle all the interactive jobs in the workload. *Glidein + BF* obtains better average wait times and makespan due to the policy of starting jobs that are not at the head of the queue. The limit imposed for the number of CPUs allocated when the first job is not able to start ensures that there will not be starvation, and that wait times for those jobs will not be much higher.

Moreover, higher interactive loads could be supported with these policies due to the availability of free virtual slots on the resources. Figure 5.16 shows the average number of virtual slots created and used during a day of simulation for *Glidein + FCFS*. The entire simulation spans approximately 11 days. Half of the virtual slots are always used but there is space for more jobs even in moments of higher load. Virtual slots are only maintained on resources with a job running.

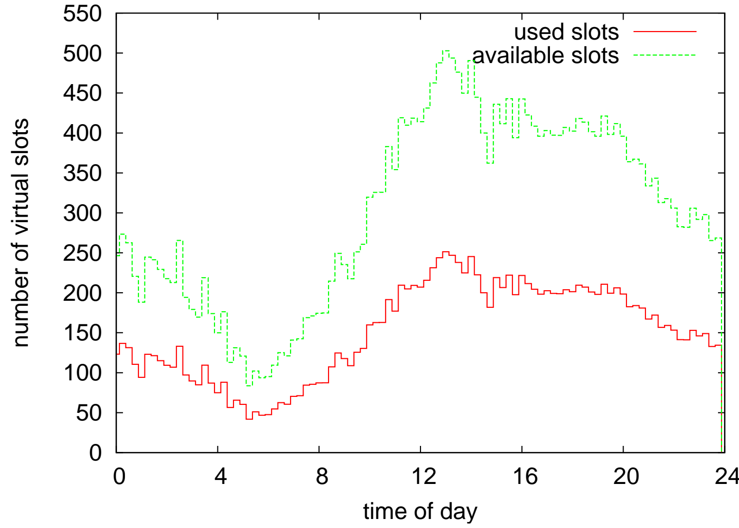


Figure 5.16: Average number of virtual slots created and used.

The average slowdowns and wait times maintain similar values for the three policies. No free CPUs always produces long wait times in queues for batch jobs. As explained in the previous section, the Glidein policies introduce extra overhead due to the creation of virtual slots, increasing the slowdown and wait times. Table 5.6 shows the average wait times, the minimum and maximum wait times, and the average slowdown for interactive jobs for the three policies.

Policy	AWT (s)	min WT (s)	max WT (s)	ASD
FCFS	233.4	229.6	251.2	1.13
Glidein + FCFS	163.2	43.4	249.0	1.11
Glidein + BF	148.4	44.1	250.3	1.10

Table 5.6: Metrics for interactive jobs.

In the *FCFS* policy, interactive jobs pass through the LRMS at the resources, resulting in Average Wait Time of 233.4 s, with small variation in the minimum and maximum wait times. For the policies with Glidein, some interactive jobs start faster due to the mechanism, and therefore the AWT decreases to 163.2 s (*Glidein+FCFS* policy) and 148.4 s (*Glidein+BF* policy). When there are free resources, the jobs do not use the Glidein mechanism, resulting in maximum wait times similar to the ones for the *FCFS* policy. However, if the mechanism is used, the wait time is greatly reduced, because the job is started directly on available virtual slots using the Condor mechanisms instead of going through the Globus middleware and LRMS queues. As seen in the Table, the minimum wait time is below 45 s for both Glidein policies.

Although the Glidein mechanism allows fast execution of interactive jobs, the batch jobs that caused initial allocation of the machine have longer execution times, because they share the CPU and have lower priority. Table 5.7 shows the impact on run time for batch jobs. The impact is measured as the result of $\frac{r_{model}-r_{real}}{r_{model}}$, where r_{model} is the run time of the job as specified by the model, and r_{real} is the run time of the job on the simulated system. The table shows the average impact for batch jobs that run without being preempted by an interactive job, and for the ones that were executed sharing the CPU.

Policy	Non Preempted jobs	σ	Preempted jobs	σ
FCFS	0.01	0.002	–	–
Glidein + FCFS	0.01	0.002	0.13	0.07
Glidein + BF	0.01	0.003	0.14	0.09

Table 5.7: Impact of Glidein on batch jobs.

The 1% impact on run time for the non preempted jobs is mainly due to the time elapsed from the actual end of the job to the detection of the job completion at the LRMS. All the policies behave similarly here, since when only one slot is used, Glidein does not significantly affect to the execution of the batch job as seen in Section 5.1.4. For preempted jobs, the overhead is below 15%. This impact is lower than that measured in Section 5.1.4 for low priority virtual slots, because interactive jobs are usually short and do not preempt the batch job for long periods of time.

This experiment has shown how the Glidein mechanism can be used to start all the interactive jobs in the workload, even for the case of high occupancy on the resources, and obtaining low waiting times.

5.4 Conclusions

In this Chapter we have presented an experimental evaluation of the CrossBroker implementation. We have shown the overhead introduced by our system. Interactive jobs can be started almost immediately with the use of the Glidein mechanism, while batch jobs are not greatly impacted by our mechanism.

We presented the use of the CrossBroker on a real testbed, where a real workload is outlined. There is a high success rate when scheduling jobs in this environment.

In order to evaluate the benefits for interactive and parallel jobs, we simulated the int.eu.grid testbed and the CrossBroker using the GridSim simulator. The workload for the simulation was generated according the Lublin model, which is largely used by the resource management research community for taking into account the char-

acteristics of grids workloads and the real workload of int.eu.grid. We found that co-scheduling permits a higher utilization of the resources. When taking into account the types of jobs, the impact on their run time is minimal. We have also shown that the Glidein mechanism allows the execution of more interactive jobs, even under high occupancy of the resources, without introducing major overhead for the remainder of the jobs.

CHAPTER 6

Conclusions and Future Research

In this work we have studied the problem of executing parallel and interactive jobs in grid environments. In this Chapter we review the main conclusions and present the new lines of research opened by this work.

Grid computing provides a large-scale, multi-organizational infrastructure where the next generation of scientific applications will run. To date, the focus in grid resource management has been on offering services for serial batch jobs. However, researchers can benefit from access to powerful resources in an interactive mode for the final stage of their analysis in a wide range of applications. This introduces the necessity of providing a channel of communication between the applications and the users, and it requires the possibility of job start immediate future, also taking into account situations in which most computing resources may be running batch jobs. Moreover, grid environments provide the possibility to run parallel applications that can efficiently use many processors, and take advantage of using sets of resources. In order to execute such applications, co-allocation of resources within different administrative domains is needed.

We proposed a new architecture that addresses the challenges of executing both interactive and parallel jobs in a grid environment. This architecture defines a job

model and a set of execution components to provide transparent and reliable services for the applications:

- The *Application Launchers* manage the grid level start up of applications and provide the co-allocation services needed for parallel applications.
- *Job Starters* are responsible for invoking the applications at the resource level, handling all the low level details of the resources, and the parallel application implementation
- The *Interactive Agents* support the on-line interaction of the application with the user in a grid environment.

Those components can be combined to support the execution of applications using a variety of parallel library implementations and the use of different interactive channel forwarding mechanisms. We also presented a job definition language to allow the specification of the jobs conforming to the proposed architecture by extending the JDL language.

The execution of parallel and interactive jobs in a grid environment requires additional mechanisms to provide fast start up and co-allocation under different administrative domains. We introduced a time-sharing mechanism that enables both interactive and batch jobs to share a single machine, in such a way that the interactive application starts its execution as soon as it is submitted. This mechanism can be also used to co-allocate the different components of a job on several resources, without having idle resources.

The proposed architecture has been implemented in the CrossBroker, a Grid Resource Management System that has been used in the production services of the CrossGrid and int.eu.grid European Projects. The implementation leverages existing efforts in the area by taking advantage of already available components. We have described the design and implementation of the system. This resource manager gives transparent support for execution of parallel applications that use Open MPI, MPICH, PACX-MPI and MPICH-G2 libraries for communication. It ensures the co-allocation of the jobs that need the libraries. It abstracts the low-level details and heterogeneity of grid environments from the user from. The CrossBroker also includes services for interactive agents with the support of the time-sharing mechanism and the Interactive Agents framework.

We performed an experimental evaluation of our system. We measured the overhead introduced by the CrossBroker and the execution environment used. The overhead introduced by our system are tolerable for the execution of interactive jobs, while it does not greatly impact batch jobs. We shown the statistics for jobs executed in the int.eu.grid project and shown two example applications that use the features of our

system. Additionally, by using simulation, we have shown the benefits of using the CrossBroker for interactive and parallel applications: interactive applications can be executed even under the high occupancy of resources and parallel applications can be executed using more resources, while avoiding fragmentation by using co-allocation.

The main contributions of this work can be found in the following publications:

CAI 2008a E. Fernández, A. Cencerrado, E. Heymann, M. A. Senar, *CrossBroker: A Grid Metascheduler for Interactive and Parallel Jobs*, Computing and Informatics, vol. 27, pp. 187–197, 2008.

CAI 2008b K. Dichev, S. Stork, R. Keller, E. Fernández, *MPI Support on the Grid*, Computing and Informatics, vol. 27, pp. 213–222, 2008

RedIris 2007 E. Fernández, A. Morajko, A. Fernández, M. A. Senar, E. Heymann, *CrossBroker: gestión de aplicaciones paralelas e interactivas en entornos Grid.* Boletín de la red nacional de I+D, RedIRIS, pp. 35–39, 2007

Cluster 2006 E. Fernández, E. Heymann, M. A. Senar, *Resource Management for Interactive Jobs in a Grid Environment.* Proceedings of the 2006 IEEE International Conference on Cluster Computing, pp: 1–10, 2006.

EuroPar 2006 E. Fernández, E. Heymann, M.A. Senar, *Supporting Efficient Execution of MPI Applications Across Multiple Sites.* Proceedings of the Euro-Par 2006 Parallel Processing, LNCS Series, vol. 4128, pp. 383–392, Springer, 2006.

EGC 2005 A. Morajko, E. Fernández, A. Fernández, E. Heymann, M. A. Senar, *Workflow Management in the CrossGrid Project.* Proceedings of the Advances in Grid Computing - European Grid Conference 2005, LNCS Series, vol. 3470, pp. 424–433, Springer 2005.

AxG 2004 E. Heymann, M. A. Senar, E. Fernández, A. Fernández, J. Salt, *The EU-CrossGrid Approach for Grid Application Scheduling.* Proceedings of the Second European AcrossGrids Conference, AxGrids 2004, LNCS Series, vol. 3165, pp. 42–50, Springer 2004.

Jornadas 2004 E. Fernández, E., Heymann, M. A. Senar, E. Luque, A. Fernández, *Reliable Scheduling of MPI Applications.* Actas de las XV Jornadas de Paralelismo (Proceedings of the XV Spanish Workshop on Parallel Computing), Almería, Spain, pp. 301–306 , 2004.

6.1 Open Lines of Research

Although the proposed architecture and the CrossBroker implementation has been tested in a production environment and offers a fully operational management of

interactive and parallel jobs, open lines of research remain to be explored.

- Our CrossBroker implementation is a centralized one, where a single global scheduler manages the jobs. While there could be more than one CrossBroker managing the same set of resources for different users, each would act independently from the others and conflicting scheduling decisions would frequently arise. Moreover, having centralized services introduces a potential bottleneck when the number of jobs and users rises. Creating a distributed global grid scheduler would avoid bottlenecks and allow the CrossBroker to scale gracefully. An extensive study of the CrossBroker scheduling policies on a real testbed should be made in order to better evaluate the scheduler.

- The high priority of interactive jobs can lead to abuse from users that take profit from the fast start up and better priority by running batch jobs. A global policy that takes into account all the jobs submitted to the system should enforce a fair-share between users. Users would have a dynamic priority, which determines how many resources they can use at a given time, considering both the batch and interactive jobs already submitted.

- The multi-programming mechanism statically assigns the job priority without any knowledge from the application. The fine adjustment of the priority for each of the jobs running on virtual slots could be accomplished on-line in an automatic way. This can be explored by future work.

- Multi-programming opens the possibility of exploring more complex scenarios, where more than two jobs are executed concurrently on the same physical machine. The allocation of each of the virtual slots for different kinds of jobs or different users with different priorities needs to be investigated.

- Although users would benefit from a reservation of resources, there is little support for this on grid resources. Since Glidein takes control of a remote machine, future work could explore the possibility of using Glidein as a general advance reservation mechanism.

Bibliography

[1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[2] I. Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[3] T. Hey and A. E. Trefethen. The UK e-Science core programme and the grid. *Future Generation Computer Systems*, 18(8):1017–1031, 2002.

[4] I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 7th International Euro-Par Conference*, pages 1–4, London (UK), 2001. Springer-Verlag.

[5] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Software: Practices and Experiences*, 32(15):1437–1466, 2002.

[6] R. Aiken and et al. Network Policy and Services: A Report of a Workshop on Middleware. RFC 2768, IETF, 2000.

[7] Open Grid Forum (OGF). Available from World Wide Web: `http://www.ogf.org/` [cited March, 2008].

[8] I. Foster and el alter. The Open Grid Services Architecture, version 1.0. Technical report, Global Grid Forum (GGF), January 2005.

[9] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[10] D. W. Erwin and D. F. Snelling. UNICORE: A Grid Computing Environment. In *Proceedings of the 7th International Euro-Par Conference*, volume 2150 of *Lecture Notes in Computer Science*, pages 825–834. Springer, 2001.

[11] OASIS WSRF Technical Commitee. OASIS WSRF v1.2 standard. Technical report, OASIS, April 2006.

[12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[13] T. A. Defanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview of the i-WAY: Wide-area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[14] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[15] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. *Job Scheduling Strategies for Parallel Processing*, 1459:62–82, 1998.

[16] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 365–375, 1997.

[17] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 181–194, August 2001.

[18] C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. RFC 2510, IETF, 1999.

[19] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[20] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable batch system: External reference specification. Technical report, MRJ Technology Solutions, November 1999.

[21] S. Zhou. Lsf: Load sharing in large-scale heterogeneous distributed systems. In *Proccedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992.

[22] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[23] W. Gentzsch. Sun grid engine: towards creating a compute power grid. In *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[24] V. Koutsonikola and A. Vakali. Ldap: framework, practices, and trends. *Internet Computing, IEEE*, 8(5):66–72, 2004.

[25] Eurogrid Project. Available from World Wide Web: `http://www.eurogrid.org/` [cited March, 2008].

[26] Vertically Integrated Optical Testbed for Large Applications (VIOLA). Available from World Wide Web: `http://www.viola-testbed.de/` [cited March, 2008].

[27] National Research Grid Initiative (NAREGI), Japan. Available from World Wide Web: `http://www.naregi.org/index_e.html` [cited March, 2008].

[28] European Data Grid (EDG) project. Available from World Wide Web: `http://eu-.web.cern.ch/` [cited March, 2008].

[29] F. Gagliardi, B. Jones, M. Reale, and S. Burke. *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459/2002 of *Lecture Notes in Computer Science*, chapter European DataGrid Project: Experiences of Deploying a Large Scale Testbed for E-science Applications, pages 255–264. Springer, 2002.

[30] H. Stockinger, F. Donno, E. Laure, S. Muzaffar, P. Kunszt, G. Andronico, and P. Millar. Grid data management in action: Experience in running and supporting data management services in the eu datagrid project. In *Proceedings of the International Computing in High Energy and Nuclear Physics (CHEP 2003)*, La Jolla, CA, March 2003.

[31] S. Andreozzi and el alter. GLUE Schema Specification - version 1.2. Technical report, Open Grid Forum, 2005.

[32] J.-P. Baud, J. Casey, S. Lemaitre, and C. Nicholson. Performance analysis of a file catalog for the LHC computing grid. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 91–99, July 2005.

[33] A. Cooke and et alter. R-GMA: An Information Integration System for Grid Monitoring. In *Proceedings of the On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 462–481, 2003.

[34] Large Hadron Collider (LHC). Available from World Wide Web: http://lhc.web.cern.ch/lhc/ [cited March, 2008].

[35] Open Science Grid (OSG). Available from World Wide Web: http://www.opensciencegrid.org/ [cited March, 2008].

[36] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 7–9, 2001.

[37] Distributed European Infrastructure for Supercomputing Applications (DEISA). Available from World Wide Web: www.deisa.org/ [cited March, 2008].

[38] Geánt network. Available from World Wide Web: http://www.geant.net [cited March, 2008].

[39] Crossgrid EU project. Available from World Wide Web: http://www.crossgrid.org/ [cited March, 2008].

[40] Interactive European Grid project. Available from World Wide Web: http://www.interactive-grid.eu/ [cited March, 2008].

[41] M. Kupczyk, R. Lichwala, N. Meyer, B. Palak, M. Plóciennik, and P. Wolniewicz. "Applications on demand" as the exploitation of the Migrating Desktop. *Future Generation Computer Systems*, 21(1):37–44, 2005.

[42] C. Anglano and et alter. Integrating grid tools to build a computing resource broker: Activities of datagrid WP1. In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP 2001)*, December 2001.

[43] Message Passing Interface Forum. MPI: A Message Passing Interface standard, June 1995.

[44] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[45] N. T. Karonis and et alters. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[46] J. Gomes and et alter. First prototype of the crossgrid testbed. In *Proceedings of the 1st European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 67–77. Springer, 2004.

[47] J. Gomes and et alter. Experience with the international testbed in the crossgrid project. In *Proceedings of the 1st European Grid Conference*, volume 3470 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2005.

[48] J. Marco and et alter. The interactive european grid: Project objectives and achievements. *Computing and Informatics*, 27(2):161–171, 2008.

[49] K. Dichev, S. Stork, R. Keller, and E. Fernández. Mpi support on the grid. *Computing and Informatics*, 27(3):213–222, 2008.

[50] H. Rosmanith and J. Volkert. Interactive techniques in grid computing: A survey. *Computing and Informatics*, 27:199–211, 2008.

[51] E. Gabriel and et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the* 11th *European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 2004.

[52] M. Polak and D. Kranzlmüller. Interactive videostreaming visualization on grids. *Future Generation Computer Systems*, 24(1):39–45, January 2008.

[53] J. Gomes and et alter. A grid infrastructure for parallel and interactive applications. *Computing and Informatics*, 27(2):173–185, 2008.

[54] J. M. Schopf. *Grid resource management: state of the art and future trends*, chapter Ten actions when Grid scheduling: the user as a Grid scheduler, pages 15–23. Kluwer Academic Publishers, 2004.

[55] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[56] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region*, volume 1, pages 283–289. IEEE Computer Society, 2000.

[57] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14:1507–1542, 2002.

[58] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw. The legion resource management system. In *Proceedings of the 12th Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 162–178, London, UK, 1999. Springer-Verlag.

[59] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *Proceedings of the 2nd workshop on Middleware for grid computing (MGC'04)*, pages 75–80, 2004.

[60] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, may 1997.

[61] K. Cooper and et alter. New grid scheduling and rescheduling methods in the grads project. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 199–207, April 2004.

[62] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software Practice & Experience*, 34(7):631–651, 2004.

[63] E. Huedo, R. S. Montero, and I. M. Llorente. The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing – Practice and Experience*, 6(3):1–8, 2005.

[64] P. Andreetto and et alter. Practical approaches to grid workload and resource management in the EGEE project. In *Proceedings of the International Computing in High Energy and Nuclear Physics (CHEP 2004)*, page 4, 2004.

[65] G. Avellino and et alter. The DataGrid Workload Management System: Challenges and results. *Journal of Grid Computing*, 2(4):353–367, December 2004.

[66] H.H. Mohamed and D.H.J. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, volume 2, pages 784–791, May 2005.

[67] H.H. Mohamed and D.H.J. Epema. The design and implementation of the koala co-allocating grid scheduler. In *Proceedings of Advances in Grid Computing - EGC 2005*, pages 640–650, 2005.

[68] The distributed ASCI supercomputer (DAS). Available from World Wide Web: `http://www.cs.vu.nl/das2` [cited March, 2008].

[69] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On advantages of grid computing for parallel job scheduling. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, page 39, May 2002.

[70] A. I. D. Bucur and D. H. J. Epema. The performance of processor co-allocation in multicluster systems. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, page 414, May 2003.

[71] H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, pages 784–790. IEEE Computer Society, 1998.

[72] E. Argollo, A. Gaudiani, D. Rexachs, and E. Luque. Tuning application in a multi-cluster environment. In *Proceedings of the 12th International Euro-Par Conference*, volume 4128 of *Lecture Notes in Computer Science*, pages 78–88. Springer, 2006.

[73] MPI Forum. Available from World Wide Web: `http://www.mpi-forum.org/` [cited March, 2008].

[74] J. M. Squyres. A component architecture for lam/mpi). In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 379–387, New York, NY, USA, 2003. ACM.

[75] R. L. Graham and et alter. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4):285–303, 2003.

[76] G. Fagg and J. Dongarra. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1908/2000 of *Lecture Notes in Computer Science*, chapter FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World, pages 346–353. Springer, 2000.

[77] R. Keller, E. Gabriel, B. Krammer, M. S. Müller, and M. M. Resch. Towards efficient execution of mpi applications on the grid: Porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003.

[78] M. Matsuda, T. Kudoh, Y. Kodama, R. Takano, and Y. Ishikawa. Tcp adaptation for mpi on long-and-fat networks. In *Proceedings of the IEEE International Symposium on Cluster Computing*, pages 1–10, 2005.

[79] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. F. H. Hofman, and K. Verstoep. Programming environments for high-performance grid computing: the albatross project. *Future Generation Computer Systems*, 18(8):1113–1125, 2002.

[80] P. M. A. Sloot, A. Tirado-Ramos, A. G. Hoekstra, and M. Bubak. An interactive grid for non-invasive vascular reconstruction. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 309–319. IEEE Computer Society, April 2004.

[81] L. Hluchy, V.D. Tran, O. Habala, B. Simo, E. Gatial, J. Astalos, and M. Dobrucky. Flood forecasting in crossgrid project. In *Proceedings of the 2nd European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 51–60. Springer, 2004.

[82] F. Castejón, J.M. Reynolds, F. Serrano, R. Valles, A. Tarancón, and J.L. Velasco. Fusion plasma simulation in the interactive grid. *Computing and Informatics*, 27(2):261–270, 2008.

[83] M. Hardt, Seymour K., J. Dongarra, M. Zapf, and N. V. Ruiter. Interactive grid-access usign gridsolve and giggle. *Computing and Informatics*, 27(2):233–248, 2008.

[84] T. Ylönen. Ssh: secure login connections over the internet. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 37–42, Berkeley, CA, USA, July 1996. USENIX Association.

[85] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[86] C. Philips, V. Welch, and S. Wilkinson. GSI-enabled open ssh, January 2002. Available from World Wide Web: `http://grid.ncsa.uiuc.edu/ssh/` [cited March, 2008].

[87] H. Rosmanith, D. Kranzlmüller, and J. Volkert. An interactive job manager for globus. In *Proceedings of Computer Aided Systems Theory – EUROCAST 2007*, pages 431–442, 2007.

[88] H. A. Lagar-Cavilla, N. Tolia, E. de Lara, M. Satyanarayanan, and D. R. O'Hallaron. Interactive resource-intensive applications made easy. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, volume 4834 of *Lecture Notes in Computer Science*, pages 143–163. Springer, 26-30 November 2007.

[89] V. Talwar, S. Basu, and R. Kumar. Architecture and environment for enabling interactive grids. *Journal of Grid Computing*, 1(3):231–251, September 2003.

[90] W. E. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The globus striped gridftp framework and server. In *Proceedings of the ACM/IEEE Conference on Supercomputing High Performance Networking and Computing, CD-Rom*, page 54. IEEE Computer Society, November 2005.

[91] J-P. Baud and et alter. CASTOR status and evolution. In *Proceedings of the International Computing in High Energy and Nuclear Physics (CHEP 2003)*, La Jolla, CA, March 2003.

[92] F. Pacini and A. Maraschini. Job Description Language (JDL) attributes specification. Technical Report 590869, EGEE Consortium, February 2006.

[93] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th*

IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, IL, July 1998.

[94] H. Rosmanith and J. Volkert. glogin - interactive connectivity for the grid. In *Proceedings of Distributed and Parallel Systems: Cluster and Grid Computing (DAPSYS), Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 3–12, september 2004.

[95] L. Matyska and et alter. Job tracking on a grid - the logging and bookkeeping and job provenance services. Technical Report 9/2007, CESNET, 2007.

[96] Condor Team. DAGMan (Directed Acyclic Graph Manager). Available from World Wide Web: `http://www.cs.wisc.edu/dagman/` [cited March, 2008].

[97] D. Thain. Identity boxing: A new technique for consistent global identity. In *Proceedings of the ACM/IEEE Supercomputing 2005 Conference on High Performance Networking and Computing, CD-Rom*, page 51. IEEE Computer Society, November 2005.

[98] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[99] S. Son and M. Livny. Recovering internet symmetry in distributed computing. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 542–549, May 2003.

[100] Berkeley Database Information Index (BDII). Available from World Wide Web: `http://www-it.desy.de/physics/projects/grid/testbed/EDG/BDII.html` [cited March, 2008].

[101] J-P. Baud, J. Casey, S. Lemaitre, C. Nicholson, D. Smith, and G. Stewart. LCG Data Management: From EDG to EGEE. In *Proceedings of the UK eScience All Hands Meeting*, Nottingham, UK, 2005.

[102] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 5:39–47, 2001.

[103] H. Rosmanith, J. Volkert, R. Valles, F. Serrano, M. Plociennik, and M. Owsiak. Interactive fusion simulation and visualisation on the grid. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 20, Washington, DC, USA, 2007. IEEE Computer Society.

[104] A. Gutiérrez and et alters. Parallelization of a neural net training program in a grid environment. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 258–265, February 2004.

[105] D. Scardaci and G. Scuderi. A secure storage service for the glite middleware. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 261–266, Aug. 2007.

[106] The Parallel Workloads Archive (PWA). Available from World Wide Web: `http://www.cs.huji.ac.il/labs/parallel/workload` [cited June, 2008].

[107] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H.J. Epema. The grid workloads archive. *Future Generation Computer Systems*, 24(7):672–686, July 2008.

[108] C. Ernemann, B. Song, and R. Yahyapour. Scaling of workload traces. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862, pages 166–182. Springer Verlag, 2003. Lecture Notes in Computer Science.

[109] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal Parallel & Distributed Computing*, 63(11):1105–1122, November 2003.

[110] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in global computing systems. *International Journal of High Performance Computing Applications*, 14(3):268–279, 2000.

[111] A. Legrand, L. Marchal, and H. Casanova. Scheduling distributed applications: the simgrid simulation framework. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 138–145. IEEE Computer Society, May 2003.

[112] R. Buyya and M. M. Murshed. Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.

[113] A. Sulistio, G. Poduval, R. Buyya, and C.-K. Tham. On incorporating differentiated levels of network service into gridsim. *Future Generation Computer Systems*, 23(4):606–615, 2007.

[114] J. L. Albín, J. A. Lorenzo, J. C. Cabaleiro, T. F. Pena, and F. F. Rivera. Simulation of parallel applications in gridsim. In *Proceedings of the Iberian Grid Infrastructure Conference*, pages 208–219, May 2007.