UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS
DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

An Approach to Ease the Gridification of Conventional Applications

by
Cristian Maximiliano Mateos Diaz

A thesis report submitted in partial fullfilment
of the requirements for the degree of
Doctor in Computer Science

Dr. Marcelo Campo
Advisor
Dr. Alejandro Zunino
Co-advisor

Tandil, October 2007

UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS
DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

Un Enfoque para Facilitar la Gridificación de Aplicaciones Convencionales

por
Cristian Maximiliano Mateos Diaz

Tesis sometida a evaluación como requisito parcial
para la obtención del grado de
Doctor en Ciencias de la Computación

Dr. Marcelo Campo
Director
Dr. Alejandro Zunino
Co-director

Tandil, Octubre de 2007

Abstract

The Grid shows itself as a globally distributed computing environment in which hardware and software resources from disparate sites are virtualized to provide applications with vast capabilities through a myriad of services. Just like an electrical infrastructure, which usually spreads over several cities to transparently convey and deliver electricity, the Grid aims at offering a powerful yet easy-to-use computing infrastructure to which applications can be easily "plugged" and efficiently executed by leveraging the various resources hosted at different administration domains. In fact, the term "Grid" comes from a widely-known analogy with the electrical power grid infrastructure, since researchers expect that applications will benefit from Grid resources as transparently and pervasively as electricity is now consumed by users.

The extremely heterogeneous, complex nature inherent to the Grid has now put on the agenda of the Grid research community the need for new techniques to easily gridify software that has not been at first thought to be deployed on Grid settings. Examples of this kind of software include desktop applications, legacy code and standard Web applications. As a consequence, Grid researchers are currently investigating appropriate ways of transforming non-Grid applications to effectively and efficiently benefit from Grid services. In this sense, a number of methods and tools have been recently proposed in a serious attempt to help reality to catch up with the ambitious goal of porting applications to the Grid with very little or zero effort.

However, the sad part of the story is that plugging applications to the Grid is still very difficult, since current toolkits for Grid development force developers to take into account many details when adapting applications to run on the Grid. On one hand, toolkits demands the developer to manually modify and restructure applications so they can use a particular programming API for accessing Grid services or are compliant to a certain application structure. As a consequence, benefiting from the Grid is just circumscribed to application developers having experience and knowledge on the art of Grid programming. Similarly, toolkits either lack of proper mechanisms to tune their applications once these are ported to the Grid, or provide tuning mechanisms that require solid knowledge on the application execution support being used. On the other hand, many approaches are based on low-level, non-widely employed technologies. This fact clearly compromises cross-platform interoperability of Grid applications, which in turn represents a step back in the adoption of the Grid Computing paradigm as the basis for global collaboration.

This work describes a novel approach called GRATIS that addresses these problems in the context of service-oriented Grids. Current middlewares and toolkits demand the developer to know in advance many platform-related details before an application can efficiently take advantage of Grid services and applications. Consequently, the main goal of GRATIS is to make the task of porting applications to the Grid easier and, at the same time, provide facilities to easily deal with efficiency issues when gridifying applications. Therefore, the thesis outlined in this document shows that it is possible to code Grid applications by keeping oneself away from most Grid-related details, and effectively tune these applications without knowing too much about Grid technologies and still achieving acceptable levels of performance. In order to achieve these goals, the proposed solution adopts a non-intrusive approach to gridification promoting separation of concerns between application logic and Grid behavior, this latter representing access to Grid services and functionality for efficiently running gridified applications. The approach is focused on reducing the effort to add Grid concerns to the application logic.

To experimentally validate GRATIS, some applications were developed and comparisons with other tools for gridifying applications were performed. The experiments show that GRATIS effectively reduces the gridification effort, produces Grid-enabled applications that are isolated from Grid-related functionality, and allows applications to achieve levels of efficiency that are very competitive with respect to existing approaches.

Keywords: Grid Computing, gridification, service-oriented Grids, dependency injection

Resumen

La Grid es un ambiente de computación masivamente distribuido en donde los recursos de hardware y software se virtualizan con el fin de dotar a las aplicaciones de vastas capacidades de ejecución a través de una infinidad de servicios. Al igual que las infraestructuras eléctricas, que a menudo se expanden a través de varias ciudades para transportar electricidad de forma transparente, la Grid apunta a ofrecer una infraestructura de computación poderosa pero fácil de usar a la cual puedan "enchufarse" aplicaciones, haciendo éstas uso de los recursos disponibles en diferentes dominios administrativos para ejecutar eficientemente. De hecho, el término "Grid" tiene precisamente su origen a partir de una analogía con la infraestructura de energía eléctrica (en inglés conocida como "the power grid"), dado que los investigadores esperan que en un futuro las aplicaciones se beneficien de los recursos de la Grid de forma tan transparente y ubicua como las personas consumen hoy en día la corriente eléctrica.

La naturaleza extremadamente heterogénea y compleja de la Grid ha motivado a la comunidad científica a destinar esfuerzos al desarrollo de nuevas técnicas que permitan de forma fácil gridificar software que no ha sido iniciamente pensado para ejecutar en ambientes Grid. Ejemplos de este tipo de software son las aplicaciones de escritorio, código legado y aplicaciones Web. En consecuencia, muchos de los investigadores del área de Grid Computing se encuentran abocados a la búsqueda de métodos apropiados para transformar aplicaciones convencionales a aplicaciones que sean capaces de utilizar los servicios de la Grid de forma eficiente. En este sentido, se han propuesto recientemente un gran número de métodos y herramientas que esencialmente buscan alcanzar el ambicioso objetivo de portar aplicaciones a la Grid con muy poco o incluso ningún esfuerzo.

Sin embargo, la parte triste de la historia es que enchufar aplicaciones a la Grid es aún muy difícil, dado que los toolkits propuestos para desarrollo Grid fuerzan a los usuarios a tener en cuenta muchos detalles al momento de adaptar aplicaciones para ejecutar en la Grid. Por un lado, los toolkits requieren la modificación y/o reestructuración manual de las aplicaciones para usar una API de programación particular de acceso a servicios Grid o responder a una estructura de aplicación predeterminada. A raíz de esto, el beneficiarse de la Grid queda circunscripto sólo a aquellos desarrolladores que poseen experiencia y conocimiento en programación Grid. Similarmente, dichos toolkits o bien carecen de mecanismos para mejorar la performance de

las aplicaciones gridificadas, o proveen mecanismos que requieren un conocimiento sólido en el soporte de tiempo de ejecución Grid por parte del usuario. Por otra parte, muchos enfoques se basan en tecnologías propietarias y no estándares. Esto condiciona enormemente la interoperabilidad entre diferentes aplicaciones y plataformas Grid, lo que a su vez representa un paso hacia atrás en la adopción del paradigma de Grid Computing como soporte a las necesidades de la colaboración global.

El presente trabajo describe un enfoque denominado GRATIS para resolver los problemas mencionados anteriormente en el contexto de las Grids orientadas a servicios. Las plataformas y toolkits actuales requieren que los desarrolladores conozcan de antemano muchos detalles relacionados con el soporte ejecución antes de poder aprovechar efectivamente los servicios de la Grid. En consecuencia, el principal objetivo de GRATIS es facilitar la tarea de adaptación de aplicaciones a la Grid y, al mismo tiempo, proveer mecanismos para hacer que las aplicaciones gridificadas ejecuten de forma eficiente. En tal sentido, la tesis descrita a lo largo de este documento muestra que es posible codificar aplicaciones Grid aislándose de la mayoría de los detalles relacionados con la puesta en marcha de una aplicación Grid, y adaptar dichas aplicaciones sin conocer demasiado acerca de la estructura interna de la Grid y aún así alcanzar niveles aceptables de performance. Para lograr los objetivos planteados, la solución propuesta adopta un enfoque de gridificación no intrusivo, es decir, promoviendo la separación entre la lógica de aplicación y el comportamiento Grid, éste último representando tanto acceso a servicios Grid como también funcionalidad para ejecución eficiente de aplicaciones gridificadas. En dicho sentido, el enfoque apunta a reducir el esfuerzo de adicionar comportamiento Grid a la lógica de aplicación.

Para validar experimentalmente GRATIS, se desarrollaron algunas aplicaciones y se llevaron a cabo comparaciones con otras herramientas para la gridificación de aplicaciones. De los experimentos surje que GRATIS reduce de forma efectiva el esfuerzo de gridificación, produce aplicaciones Grid cuyo código está aislado de la funcionalidad propia de la Grid, y permite a las aplicaciones alcanzar niveles de performance muy competitivos con respecto a los logrados mediante enfoques existentes.

Palabras clave: Grid Computing, gridificación, Grids orientadas a servicios, inyección de dependencias

Acknowledgements

There are many people I would like to thank for supporting and helping me over the course of this work. Without them, the present thesis would not have been possible.

First at all, I am deeply grateful to my advisors Marcelo Campo and Alejandro Zunino for their guidance, encouragement and patience, and for sharing with me the experience and knowledge they have about how to do research. I also thank Marcelo for their wisdom and their priceless advising that greatly helped me during not only my academic formation but also in some of the most challenging parts of my life.

I sincerely thank the National Council of Scientific and Technical Research (CONICET) for the financial support for this work.

I thank all the members of the ISISTAN for their valuable help, suggestions and thoughts. I also thank the undergraduate students who contributed in the implementation and evaluation of the software, including Esteban Pamio, Juan Manuel Rodriguez, Franco Scavuzzo and Guillermo Zunino.

I also want to thank Andrea and our little daughter Bianca, and my parents, for the unconditional love they gave over all these years, and for relentlessly help me keep myself a little in this world, at least from time to time, during the process of writing this thesis.

Cristian Mateos Diaz

Contents

List of Figures

Table Index

List of Algorithms

Acronyms

AMWAT AppLeS Master-Worker Application Template

AOP Aspect-Oriented Programming

API Application Program Interface

B2B Business-to-Business

CCA Common Component Architecture

CERN European Organization for Nuclear Research

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CRS Cluster-aware Random Stealing

CoG Commodity Grid

DI Dependency Injection

DS Data Service

FTP File Transfer Protocol

GAF Grid Application Framework

GAT Grid Application Toolkit

GEMLCA Grid Execution Management for Legacy Code Architecture

GNS GriddLeS Name Server

GRAM Grid Resource Allocation Manager

GRASG Gridifying and Running Applications on Service-oriented Grids

GRATIS GRidifying Applications by Transparent Injection of Services

GRIM Generalized Reactive Intelligent Mobility

GrADS Grid Application Development Software

GriddLeS Grid Enabling Legacy Software

HTML Hyper Text Markup Language

HTTP Hyper Text Transfer Protocol

HTTPS Secure HTTP

IDL Interface Definition Language

IP Internet Protocol

IS Information Service

JES Job Execution Service

JVM Java Virtual Machine

MGS Mobile Grid Services

MPI Message Passing Interface

MW Master-Worker

NAICS North American Industry Classification System

NFS Network File System

NTP Network Time Protocol

NWS Network Weather Service

LCID Legacy Code Interface Description

OGSA Open Grid Services Architecture

P2P Peer to Peer

PC Personal Computer

PDA Personal Digital Assistant

PNS Protocol Name Servers

PVM Parallel Virtual Machine

QoS Quality of Service

RASS Resource Allocation and Scheduling Service

RMI Remote Method Invocation

RMF Reactive Mobility by Failure

RPC Remote Procedure Call

SAGA Simple API for Grid Applications

SETI Search for Extraterrestial Intelligence

SIDL Scientific Interface Definition Language

SMTP Simple Mail Transfer Protocol

SOA Service-Oriented Architecture

SOAP Service-Oriented Architecture Protocol

SRS Stop Restart Software

TCP Transport Control Protocol

UDDI Universal Description, Discovery, and Integration

UML Unified Modeling Language

UNSPSC Universal Standard Products and Services Codes

URI Uniform Resource Identifier

URL Uniform Resource Locator

VO Virtual Organization

VPN Virtual Private Network

W3C World Wide Web

WSDL Web Service Definition Language

WSIF Web Services Invocation Framework

WSRF Web Services Resource Framework

XML Extended Markup Language

Chapter 1

Introduction

Grid Computing is a new paradigm for distributed computing that is based on the idea of arranging the hardware and software resources of computers in a network to execute complex applications. Typically, these applications are intended to solve many scientific or technical problems that require by nature a large number of resources, such as CPU cycles and memory, network bandwidth, data, applications, services, and so on. Such an arrangement of distributed resources is usually called a *Grid* (Foster, 2003).

Grids are essentially pervasive computing environments whose objective is to provide secure and coordinated computational resource sharing between administrative boundaries. The widely accepted definition for the notion of "Grid" was first proposed in (Foster, 2002), and states that a Grid is a system that (a) "coordinates resources that are not subject to centralized control" by bringing together resources from different control domains, (b) "uses standard, open, general-purpose protocols and interfaces" to effectively address issues such as authentication, authorization, resource discovery, and resource access, and (c) "delivers nontrivial qualities of service" by combining resources from distributed sources to meet complex user demands.

Looking back into the history, the first attempts to establish Grid settings were focused on providing infrastructures to support compute-intensive, large-scale applications by linking supercomputers (Foster et al., 2001). At that time, a Grid was conceived as a network linking together a number of very powerful, dedicated computers to harness their total processing capabilities. With the inception of Internet standards such as Internet Protocol (IP) and Transport Control Protocol (TCP) in the 1990s, and the great popularity they gained thereafter, a new phase of the evolution of Grids known as Volunteer Computing (Sarmenta and Hirano, 1999) was born. Basically, Volunteer Computing promotes the creation of Internet-wide Grids in which thousands of people share the unused processor cycles of their desktop computers.

A good example of such a system is the SETI@home project (Anderson et al., 2002), where users can sign up to have their PC process radio signals from the outer space, thus helping in a global search for the existence of life forms elsewhere in the universe. A portion of the signal data is sent from a server to a computer over the Internet and then locally processed by

a software that executes when the computer is idle (i.e. running the screensaver). Finally, the results are sent back to the SETI@home servers. Other examples of initiatives by which users around the world can donate CPU power to academic research and public-interest projects are Distributed.NET (Distributed.net, 1997) and Folding@home (Folding@home, 2000).

A more contemporary, service-oriented view of Grids came into existence with the introduction of the first Grid runtime systems. Examples of popular platforms for Grid application development and execution are Legion (Natrajan et al., 2002), Condor (Thain et al., 2003) and Globus (Foster, 2005). In short, the aim of these platforms is to provide developers with an Application Program Interface (API) implementing services by which Grid resources can be accessed and efficiently consumed from within applications. Meanwhile, many well-established standardization forums such as the Open Grid Forum and the OASIS Consortium were producing the first global standards for the Grid. Results of these efforts include Open Grid Services Architecture (OGSA) (OGSA-WG, 2005), a service-oriented Grid reference architecture, and Web Services Resource Framework (WSRF) (OASIS Consortium, 2006), a standard for describing and consuming Grid resources by means of Web Service technologies (Vaughan-Nichols, 2002).

Nowadays, research in the Grid community is tending to strongly follow the vision that originally gave birth to the term "Grid" (Stockinger, 2007). Basically, the term was originated from an analogy with the electrical power grid, because the long-term goal of Grid Computing is, besides virtualizing resources and providing powerful computational environments, letting applications access Grid resources as easy and pervasively as electrical power is now consumed by appliances from wall sockets (Chetty and Buyya, 2002; Taylor, 2005). In other words, the idea is to provide applications with much better execution capabilities, and at the same time to hide the inherent complexity of Grid infrastructures as much as possible from the application developer.

From a software development perspective, a major goal of Grid Computing is to allow programmers to easily code applications (i.e. the "appliances") and deploy them on a Grid setting (i.e. "plug them"), thus delegating to the Grid the responsibility for locating and utilizing the resources that are necessary to efficiently execute applications. Even much more interesting, it would be better to allow *any* existing application to be straightforwardly plugged onto the Grid. Unfortunately, the envisioned analogy between computer Grids and electrical power grids does not fully hold yet. The reason for this to happen is that porting or "gridifying" an application still requires the developer to be actively involved in rewriting and restructuring the application code and/or performing deployment tasks in order to Grid-enabling the application. Therefore, the use of Grids remains circumscribed only to those users and programmers who have a solid knowledge on Grid technologies.

1.1 Motivation

Unlike the electrical power grid, which can be easily used in a plug and play fashion, the Grid is very complex to use (Chetty and Buyya, 2002). Certainly, plugging applications to effortlessly

benefit from Grid resources is still far from being as easy as connecting a toaster, a television or even a personal computer to the electrical power grid. The cause for which this has been historically so difficult is twofold:

- **Interfacing the Grid by means of programming APIs.** Many Grid toolkits are mostly oriented towards providing APIs for accessing Grid resources and implementing Grid applications from scratch. As a consequence, the application logic turns out to be mixed up with API code for using Grid services, thus making maintainability, out-of-the-box testing, legibility, and portability to different Grid platforms very hard. Besides, these toolkits not only require developers to adapt their applications to the API being used, but also force them to learn yet another programming API. This hinders the adoption of these technologies by novice Grid developers. A number of toolkits following this approach can be found in the literature, such as the Java Commodity Grid (CoG) Kit (Globus Alliance, 2006), the Grid Application Framework (GAF) (IBM alphaWorks, 2004) and the Simple API for Grid Applications (SAGA)(Goodale et al., 2006), just to name a few.

  Moreover, integrating legacy source code onto the Grid demands in many cases to rewrite/-restructure significant portions of the application in order to use Grid APIs. This problem is partially addressed by tools that take applications in their binary form, along with some user-provided configuration (e.g., input/output parameters and resource requirements), and wrap the executable code with a software entity (e.g. a Web Service) that isolates the complex details of the underlying Grid. Examples of such tools are GriddLes (Kommineni and Abramson, 2005), GEMLCA (Delaittre et al., 2005) and GRASG (Ho et al., 2006). However, this approach results in extremely coarse-grained Grid applications, thus users generally cannot control the execution of their applications in a fine-grained manner to make better use of Grid resources, such as parallelizing certain parts of an application or distributing the execution of single application components. Overall, this represents a clear tradeoff between ease of gridification versus flexibility to configure the runtime aspects of a gridified application.

- **Using programming models that call for application restructuring.** Many Grid technologies assume a rather hard-to-use programming model such as message passing or remote procedure call such as MPICH-G2 (Karonis et al., 2003) and GridRPC (Nakada et al., 2005). Similarly, Grid frameworks such as Master-Worker (MW) (Goux et al., 2000) and JaSkel (Ferreira et al., 2006) prescribe programming models based on templates, that is, recurring design patterns commonly found in Grid applications. Though appropriate for taking advantage of distributed and parallel execution, applications developed under these models must be manually partitioned into individual communicating components. The developer is responsible for explicitly managing non-functional aspects of the application such as parallelism, coordination and location of these components. As a consequence, the developer is not just focused on implementing the logic of his application, but also on coding/adapting it in such a way it is compliant to the framework being used. Note

that, depending on the complexity of the framework, this may be a difficult task for unskilled developers and may require a significant amount of redesign when gridifying existing applications.

On the other hand, application components resulted from using these programming models usually talk through ad-hoc, non-interoperable protocols and mechanisms. This fact limits the potential of these solutions as the basis for future Grid technology, which clearly will need to be highly interoperable to meet the needs of global enterprises. Here, Web Services technologies play a fundamental role (Atkinson et al., 2005), since they help in providing a satisfactory solution to the problem of heterogeneous systems integration, which will be a fundamental requirement of almost every Grid-based application in the near future.

Approaches falling in either of the two above categories share a common goal: they are focused on offering *programming* abstractions rather than *gridification* facilities. Broadly, "gridification" can be defined as the process of transforming an ordinary application to run on the Grid (Mateos et al., 2007a). An ordinary application is a software that has not been at first thought to be deployed on Grid settings, such as desktop applications or legacy code. In order to perform gridification, it is clear that a conventional application that a user may have written needs to be adapted (configured, modified or restructured) in order to access and use Grid resources. This is similar –to a certain extent– to the process of adapting an application to run as a client/-server Web application. In both cases, users have to invest some effort into transforming their applications to run in the new environment. Since the Grid is very complex, the big challenge is therefore to keep this effort as low as possible and, ideally, make it disappear.

As Grid technologies tend more and more to converge with SOAs (Service-Oriented Architecture) (Atkinson et al., 2005), the problem of gridification reduces to that of lessening the effort of seamlessly using Grid services. As a consequence, Grid researchers are currently engaged in finding appropriate ways of transforming conventional user applications to effectively and efficiently take benefit from Grid services. However, efforts towards this end are immature from the point of view of software gridification, and are inspired by the idea of "applications are coded for the Grid", which means developers must keep in mind many Grid-related implementation details when gridifying their applications. Mostly, developers are forced not only to employ programming models and middleware-level abstractions that potentially differ from the ones used in the original application, but also to manually manage the tasks of accessing and interacting with Grid services.

It is therefore crucial for gridification to become a reality that Grid technologies follow a different approach in which applications are easily adapted to run on the Grid rather than coded for the Grid. Specifically, the integrity of the application logic must be preserved, in the sense developers should only be focused first on coding the pure functional behavior of their applications, and be able to non-intrusively add Grid behavior to them afterwards (i.e. access to Grid services). Even when gridification might require users to alter the code or structure of their applications, modifications should be minimal. As will be explained, the idea is to transparently "inject"

Grid services into the application upon gridification, rather than having to explicitly alter the application logic to use external services.

1.2 The Thesis

Taking into account the problems discussed above, it is clear that there is a need for a new approach that is capable of effectively coping with these issues. In this sense, this work describes an approach whose utmost goal is to provide better support –in terms of simplicity– for gridifying ordinary applications. It is worth noting that the solution proposed in this thesis does not attempt to provide a method to gridify any kind of application, but a method oriented towards the gridification of component-based applications, that is, applications where building blocks are described by and interact through well-defined interfaces. Similarly, the solution does not target all types of Grid, but only *service-oriented* Grids, in which their capabilities are accessible through a set of services.

The approach described in this work seeks to significantly ease the task of porting software to the Grid, but without negatively affecting the performance of the resulting applications. In this sense, the main hypothesis of this dissertation is that it is possible to have little gridification effort and at the same time producing Grid-aware applications whose performance is competitive with regard to the performance levels achieved by applications gridified under existing approaches. In other words, the hypothesis states that low gridification effort and good levels of performance can coexist.

In order to materialize the approach and set the basis for experimental evaluation, a gridification tool based on the Java language have been implemented. On one hand, the component-based paradigm is a very popular programming style among Java developers. In addition, given the widely adoption of the Java language, the tool can benefit a large percentage of today's Java applications. On the other hand, Java has been widely recognized as a suitable language for distributed programming mainly because of its "write once, run anywhere" philosophy that promotes platform independence, which is a crucial feature given the extremely heterogeneous nature of Grids.

Basically, the thesis described throughout this document provides answers to the following questions:

- It is possible to add Grid-dependent behavior to a conventional application in a non--invasive way, that is, allowing developers to effectively access Grid services and external applications from within an ordinary application while keeping the requirement of modifying source code or redesigning the application very low?
- Is there a way to Grid-enable an existing application by leveraging the functionality and abstractions of existing Grid programming APIs without explicitly using these facilities in the code implementing the application logic?

- What is the appropriate standard granularity for gridified applications so they can be efficiently executed on the Grid, and how to combine this granularity with easy-to-use, non-intrusive mechanisms to manually adjust the application granularity to achieve acceptable performance levels with respect to existing approaches to gridification?

To overcome these issues, a new gridification method called GRidifying Applications by Transparent Injection of Services (GRATIS) have been developed. Central to GRATIS is the concept of Dependency Injection (DI) (Johnson, 2005), a notion similar to the Inversion of Control capability that can be found in object-oriented frameworks.

In DI, components providing certain services are transparently injected into components that require these services as needed. GRATIS exploits this concept by allowing developers to inject Grid metaservices (e.g. service discovery and invocation, mobility, parallelism, etc.) into their ordinary applications with little effort. In essence, JGRIM aims at dealing with the "ease of gridification versus flexible tuning" tradeoff discussed previously by minimizing the requirement of source code modification when porting conventional applications to the Grid, and at the same time providing easy-to-use mechanisms to effectively tune Grid applications that do not intrude on the original version of the application code.

GRATIS follows the so-called *two-step* gridification methodology (Mateos et al., 2007a), in which developers are allowed to focus first on implementing, testing and optimizing the functional code of their applications, and then to Grid-enable them. Additionally, GRATIS promotes separation between application logic and Grid behavior by non-invasively injecting all Grid-related services needed by the application at gridification time.

The next subsection presents further details on the approach to gridification described in this thesis.

1.2.1 GRATIS: Injecting Grid services into conventional applications

The goal of DI is to achieve higher levels of decoupling in component-based applications by having the components described through public interfaces, and removing dependencies between components by delegating the responsibility for component instance creation and linking to a DI *container*. In this way, components only know each other's public interfaces, and it is up to the DI container to create and set (i.e. "inject") to a client component an instance of another component implementing a certain interface upon invocations on that interface. By drawing a parallel with service-oriented software (Huhns and Singh, 2005), the former component can be thought as a client requesting services, whereas the latter as one representative or proxy of a concrete provider for these services. In this context, the container could be the runtime system that binds clients to service providers. For instance, an application component requesting a certain service may be transparently bound to a Web Service providing the necessary functional capabilities.

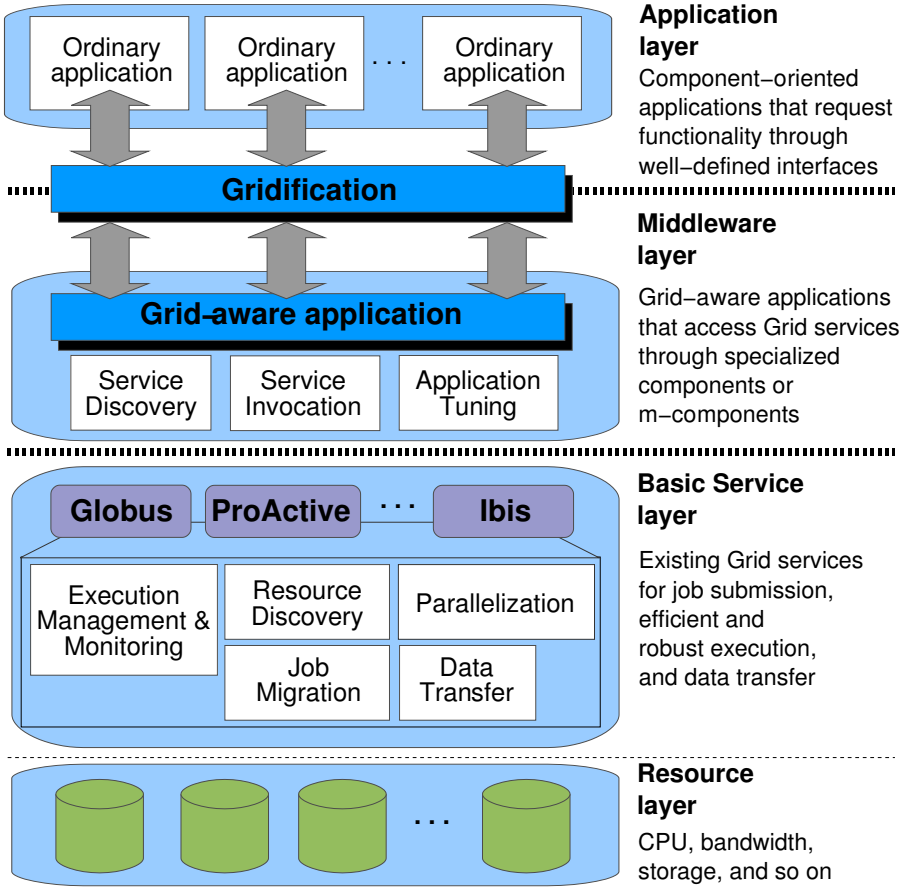


Figure 1.1: Overview of GRATIS

DI is indeed an effective way to achieve loose coupling between application components. The pattern results in very decoupled components, since the glue or boilerplate code for linking them together is not explicitly declared by the developer (Johnson, 2005). A crucial implication of the DI concept to the GRATIS approach is that dependencies can be managed by a software layer (e.g. a middleware or a runtime platform) to add Grid behavior to ordinary applications. In this sense, GRATIS essentially builds upon DI to transparently add Grid functionality such as service discovery and service invocation to conventional applications. In addition, due to the fact that performance is a fundamental requirement of most Grid applications, GRATIS also uses DI to inject tuning services such as parallelization and mobility to applications.

Figure 1.1 depicts an overview of GRATIS. As illustrated, GRATIS sits on top of existing Grid infrastructures by adding a middleware layer that enables component-based applications to seamlessly and transparently use Grid services. Basically, GRATIS can be seen as a stack composed four layers at which a different type of service provisioning is carried out:

- *Resource layer*, which just represents the physical infrastructure of a Grid, given by its resources and the necessary low-level protocols and contracts to interact with them.

- *Basic Service layer*, which provides sophisticated services to applications (e.g. load balancing, parallelization, brokering, security, etc.) by means of existing Grid platforms and resource management systems such as Globus or Condor. Basically, this is precisely the service layer that commonly serves as the Grid entry point for gridification under most of the existing approaches.

- *Middleware layer*, which hosts a number of metaservices that act as a glue between applications and the Grid. A metaservice is a representative of a set of interrelated concrete services, that is, those providing similar Grid functionality. Examples of metaservices include service and application discovery, service invocation and application tuning. Furthermore, tuning metaservices are strongly inspired on the notion of *policies* (Zunino et al., 2005a), that is, non-intrusive mechanisms by which developers can customize the way Grid services are accessed.

- *Application layer*, which is composed of component-based ordinary applications in which components are described through a clear interface to their operations. After gridification, operation requests originated at the application level are transparently handled by metaservices, which are basically in charge of dealing with service discovery and interaction within the Grid.

To validate the approach, the middleware layer mentioned above has been implemented as a Java-based toolkit called JGRIM (Mateos et al., 2008). JGRIM provides abstractions to make gridification easier, letting developers to focus on the development and testing of application logic without worrying about common Grid-related implementation details such as resource discovery, Grid service invocation and communication protocols. In other words, the goal of JGRIM is to permit applications to discover and efficiently use the vast amount of services offered by a Grid without the need to explicitly provide code for either finding or invoking these services from within the application logic.

The runtime support of JGRIM is a materialization of Generalised Reactive Intelligent Mobility (GRIM) (Mateos et al., 2005; Mateos et al., 2007b), a generic agent execution model that allows developers to easily manage mobility of agents and resources. Grid applications in JGRIM (i.e. a gridified application) are application-level mobile entities called Mobile Grid Services (MGS) which interact with other MGSs and have injected Grid services to perform mobility, execution and resource brokering. An MGS comprises the logic (what the service does), and the Grid-dependent behavior, which is configured separately from the logic and glued to the MGS at deployment time. Essentially, the gridification process of JGRIM is intended to work by semi-automatically transforming ordinary applications to MGSs.

Mobile agents have been recognized as a good alternative for materializing Grid middleware services and applications (Di Martino and Rana, 2004; Fukuda et al., 2006). Basically, a mobile agent is a software that can migrate within the nodes of a network to perform tasks by locally interacting with computer resources (Tripathi et al., 2002). Mobile agents have some properties

that make them suitable for exploiting the resources of the Grid, because they add mobility to the rest of the properties of software agents, which may potentially cause performance improvements when interacting with remote resources, and are by nature location-aware, which allows them to behave according to the execution context (e.g. available resources) where they run. Another important advantages of mobile agents is that they can work without maintaining a connection to their owner user/application, and allow developers to implement very robust and scalable distributed applications (Lange and Oshima, 1999).

1.3 Contributions

This work introduces two important contributions to the area of Grid Computing, namely:

- A novel approach to gridification called GRATIS, by which ordinary applications can be ported to a Grid setting with minimum effort, and still be able to achieve competitive levels of performance by means of policies. GRATIS combines the advantages of both component and service-oriented programming paradigms in terms of loose coupling, modularity and interoperability, and the benefits of dependency injection, which allows to furnish conventional applications with Grid services without the need for source code modification. Furthermore, GRATIS goes beyond portability of applications to a Grid, since it promotes the complete separation of application logic from the code that depends on the actual environment where applications execute and interact.

- The JGRIM middleware, which is a materialization of the GRATIS approach in the context of Java, a language that has been widely recognized as an excellent choice for implementing distributed applications. JGRIM provides several metaservices to easily discover and utilize the services offered by existing Grid middlewares and other applications, and also metaservices to improve the performance of gridified applications. JGRIM shows that it is possible to greatly simplify gridification while not negatively affecting the application performance with respect to toolkits based on a programmatic approach to gridification. Besides, as JGRIM applications are mobile entities, the middleware is also suitable to implement efficient and modular mobile Grid applications.

1.4 Organization of this work

The rest of the document is organized as described next. Chapter 2 provides a conceptual background on the Grid Computing paradigm, describes the architecture of the Grid, and discusses current standards and technologies materializing this architecture. In addition, the chapter introduces the concept of gridification, and discusses in detail how it has influenced the development of Grid technologies.

Chapter 3 describes and analyzes the most relevant related work. It includes several taxonomies in order to help in categorizing the existing approaches to gridification with respect to various important aspects of the problem. The chapter ends by presenting a detailed comparison of these approaches and highlighting their drawbacks. The contents of the chapter are for the most part derived from (Mateos et al., 2007a).

Chapter 4 presents GRidifying Applications by Transparent Injection of Services (GRATIS), an approach to gridification that allows developers to port their applications to the Grid with minimum effort, and easily address performance issues once they are gridified. The chapter is partially derived from (Mateos et al., 2008).

Chapter 5 describes Generalised Reactive Intelligent Mobility (GRIM), a conceptual framework that models common interactions between mobile applications and resources in distributed systems. This chapter is partially derived from (Mateos et al., 2007b), (Mateos et al., 2005) and (Mateos et al., 2007c). Basically, GRIM is materialized by the gridification tool that is described in Chapter 6.

Chapter 6 discusses relevant aspects related to the design and implementation of JGRIM, a GRIM-based middleware that supports the GRATIS approach for gridifying component-based Java applications. Specifically, the chapter focuses on explaining the implemented mechanisms to enable the injection of Grid services to conventional applications.

Chapter 7 details the gridification of two existing applications with JGRIM, and reports an experimental evaluation between this latter and other two platforms for Grid development named Ibis and ProActive. The goal of the experiments is to evaluate both the gridification effort and performance issues. Appendixes A and B contains the implementation code used during the experimentation. Appendixes C and D contains the data tables associated to the experiments.

Chapter 8 summarizes the contributions of the thesis to the area of Grid Computing, its limitations, and some perspectives for future research.

Chapter 2

Background

The term "Grid" was created to denote a powerful and globally distributed computing environment whose purpose is to meet the increasing demands of advanced science and engineering. Within the Grid, hardware and software resources are virtualized to transparently provide applications with vast amounts of resources. Just like the electrical power grid, the Grid aims at offering a powerful yet easy-to-use computing infrastructure to which applications can be easily "plugged" and efficiently executed.

However, given the extremely heterogeneous, complex nature inherent to the Grid, writing or adapting applications to execute on the Grid is indeed a very difficult task. So comes the challenge to provide appropriate methods to *gridify* applications, that is, semi-automatic and automatic methods for transforming conventional applications to benefit from Grid resources. Both, the evolution of the Grid as an infrastructure and the need to facilitate Grid application programming, gave origin to novel concepts and technologies.

This chapter provides an overview of the purpose and architecture of the Grid, along with the implementation technologies commonly found in today's Grid systems and applications. The next section presents the most widely accepted definition for the term "Grid", and describes its architecture. Then, Section 2.2 describes popular standards towards the materialization of this architecture in the context of service-oriented Grids. Then, Section 2.3 explores the concept of "gridification" from a technical point of view. Finally, Section 2.4 concludes the chapter.

2.1 Grid Basics: A Roadmap

The term "Grid Computing" came into daily usage about ten years ago to describe a form of distributed computing in which hardware and software resources from dispersed sites are virtualized to provide applications with a single and powerful computing infrastructure (Foster and Kesselman, 2003b). This infrastructure, known as the *Grid*[1] (Foster, 2003), is a distributed

[1] Researchers commonly speak about "the Grid" as a single entity, albeit the underlying concept can be applied to any Grid-like setting.

computing environment whose objective is to provide secure and coordinated computational resource sharing between organizations. Although simple conceptually, the idea is very difficult to deal with in practice, since it is necessary to pay attention to a lot of details when building either Grid settings or applications.

Within the Grid, the use of resources such as processing power, disk storage, applications and data, often spread across different physical locations and administrative domains, is shared and optimized through virtualization and collective management. Furthermore, sharing is necessarily highly controlled. Resource providers and consumers must define clearly and carefully what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules are usually referred as a Virtual Organization (VO) (Foster et al., 2001).

Grid infrastructures were originally intended to support compute-intensive, large-scale scientific problems and applications by linking supercomputing nodes (Foster et al., 2001). During the first half of 1990s, the inception and increasing popularity of Internet standards gave birth to an early phase of the Grid evolution later known as Volunteer Computing (Sarmenta and Hirano, 1999): users from all over the world are able to donate CPU cycles by running a free program that downloads and analyzes scientific data while their PCs are idle (e.g. when the screensaver is activated). Examples of these projects[2] are Distributed.net (Distributed.net, 1997) (Internet's first general-purpose distributed computing project), Folding@home (Folding@home, 2000) (protein folding), SETI@home (Anderson et al., 2002) (search for extraterrestrial intelligence) and, more recently, Evolution@home (evolutionary biology) (Loewe, 2007). Few years after the introduction of Volunteer Computing, the first middlewares and resource management systems for implementing Grid applications over the Internet appeared. Examples are Legion (Natrajan et al., 2002), Condor (Epema et al., 1996; Thain et al., 2003) and Globus (Foster, 2005), this latter baptized by Ian Foster as the "Linux of the Grid". The evolution of Grid technologies is depicted in Figure 2.1.

Nowadays, Grid Computing is far from only attracting the scientific community. Organizations of all types and sizes are becoming aware of the great opportunities this paradigm offers to share and exploit computational resources such as information and services. In fact, a number of projects have been actively working towards providing an infrastructure for commercial and enterprise Grids settings (Chien et al., 2003; Levine and Wirt, 2003; Sun Microsystems, 2005). Furthermore, many well-established standardization forums have produced the first global standards for the Grid. Recent results of these efforts include the OGSA (OGSA-WG, 2005), a service-oriented Grid system architecture, and the WSRF (OASIS Consortium, 2006), a framework for modeling and accessing Grid resources using Web services (Vaughan-Nichols, 2002).

[2]They are now not considered Grid systems but are classified as public distributed computing systems

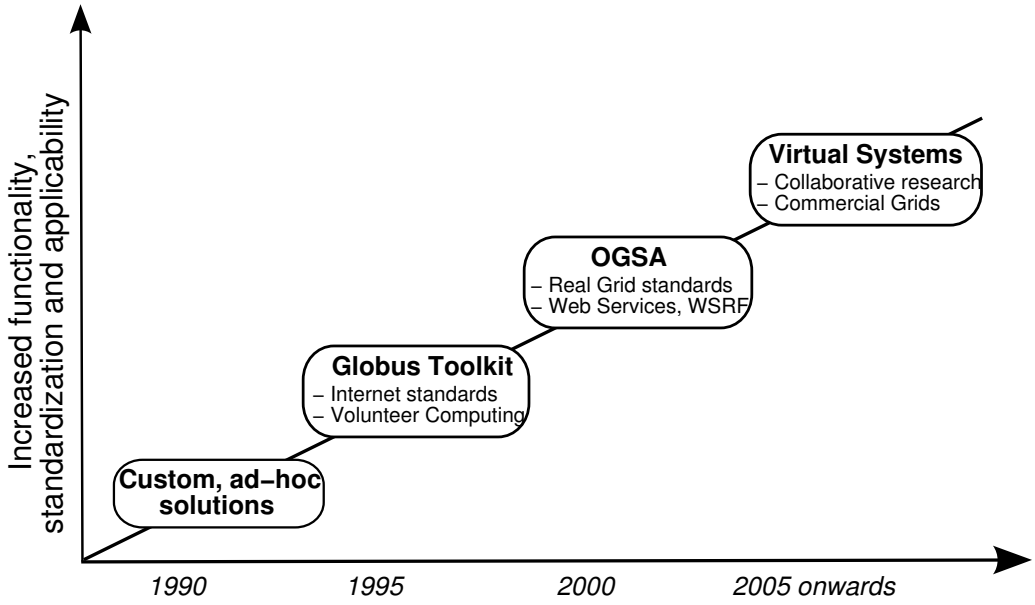


Figure 2.1: The evolution of Grid technologies (adapted from (Foster and Kesselman, 2003a))

2.1.1 The Grid: A Definition

Although many technological changes both in software and hardware have occurred since the term "Grid" was first introduced, a recent survey (Stockinger, 2007) indicates that there are hardly any significant disagreements within the Grid research community about the Grid vision. This survey was conducted by collecting the opinions of more than 170 Grid researchers from all over the world about how they define the Grid. Years ago, Ian Foster, considered by many researchers to be the father of the Grid, proposed a checklist (Foster, 2002; Foster and Kesselman, 2003a) for determining whether a system is a Grid or not, which has been broadly accepted.

According to Foster, a Grid is a system that:

- *Coordinates resources that are not subject to centralized control*, since it integrates and coordinates resources (and users) from different administrative domains by addressing issues such as security, membership, policy, and so on. For example, administrative domains may include the sales, marketing or management division of a company, the different faculties within a single university, etc. Note that if this feature is not present, the system is clearly not a Grid, but a local management system.

- *Uses standard, open, general-purpose protocols and interfaces*, since it is built upon multipurpose protocols and interfaces that address fundamental aspects such as authentication, authorization, resources discovery and access, data movement, and so forth. It is very important that these protocols and interfaces be standard and open, so as to create distributed systems that are compatible and interoperable. These principles can already be found on the Internet, where the TCP/IP protocol suite, which deals with fundamental issues like connectivity and data transmission, is composed of a set of multipurpose,

standard and open protocol specifications.

- *Delivers nontrivial qualities of service*, since it allows its software and hardware resources to be used in such a way they are capable of delivering various qualities of service (e.g. response time, throughput, reliability, robustness, etc.) to meet complex user requirements. For example, analysis of the (possibly many) petabytes of data to be produced by future high-energy physics experiments will require the coordination of hundreds of thousands of processors and hundreds of terabytes of disk space for storing intermediate results. In other words, qualities of service delivered by Grid systems are several orders of magnitude better than that of the "best" modern computer cluster we might have heard about.

Likewise, the basic Grid idea has not changed considerably within the last ten years (Stockinger, 2007). The term "Grid" comes from an analogy with the electrical power grid. Essentially, the Grid aims to let users access computational resources as transparently and pervasively as electrical power is now consumed by appliances from a wall socket (Chetty and Buyya, 2002; Taylor, 2005). Indeed, one of the goals of Grid computing is to allow software developers to build an application (i.e. "the appliance"), deploy it on the Grid (i.e. "plug it"), and then let the Grid to autonomously locate and utilize the necessary resources to execute the application. Ideally, it would be better to take any existing application and put it to work on the Grid, thus effortlessly taking advantage of Grid resources to improve performance. Sadly, the analogy does not completely hold yet since it is hard to "gridify" an application without manually rewriting or restructuring it to make it Grid-aware. Unlike the electrical power grid, which can be easily used in a plug and play fashion, the Grid is rather complex to use (Chetty and Buyya, 2002).

2.1.2 The power grid analogy

A good starting point to better understand the analogy between the Grid and the electrical power grid is GridCafé (CERN, 2007), a project from European Organization for Nuclear Research (CERN)[3] whose goal is to explain the basics of the Grid to a wider audience. Basically, GridCafé compares both infrastructures according to the three following features:

- *Transparency*: The electrical power grid is transparent because users do not know how and from where the power they use is obtained. Specifically, power is generated in numerous, geographically distributed power plants and then transparently distributed where it is needed. The Grid is also transparent, since Grid users execute applications without worrying about what kind or how many computational resources are used to perform the computations, or where all these resources are located.

- *Pervasiveness*: Electricity is available almost everywhere. The power grid is in essence an infrastructure that links together power plants by means of transmission stations, power

[3]The CERN is the world's largest particle physics laboratory, which has recently become a host for Grid Computing projects

stations, transformers and powerlines to bring electricity to homes. The Grid is also pervasive, since according to the Grid vision, computing resources and services will be accessible not only from PCs but also from laptops and mobile devices. Consequently, reusing existing pervasive infrastructures (e.g. the Internet) and ubiquitous Web technologies such as Web browsers, Java (Arnold and Gosling, 1996) and Web Services could be a big step towards complete pervasiveness and therefore easy adoption of the Grid.

- *Payment*: Grid resources are essentially utilities, since they will be provided –just like the electricity– on an on-demand and pay-per-use basis. The idea of billing users for the actual use of resources on the Grid (e.g. CPU cycles, memory and disk storage, network bandwidth, etc.) finds its roots in an old computational business model called Utility Computing, also known as On-Demand Computing (Yeo et al., 2007). A good example of a project actively working on utility-driven technologies for the Grid is Gridbus (GRIDS Laboratory, 2007).

Another feature shared by power and computational grids not mentioned in the above list is concerned with *availability*. While faults may be produced at single components of either infrastructures, it is highly desirable that those faults do not become failures (i.e. visible to users). The electrical power is (for the most part) always available even if there is a problem somewhere. Similarly, the Grid is responsible for conveying computational power wherever and whenever it is requested despite semi or complete reduction in the capabilities of its individual hardware and software components.

While the power grid infrastructure links together transmission lines and underground cables to provide users with electrical power, the Grid aims at using the Internet as the main carrier for connecting mainframes, servers and even PCs to provide scientists and application developers with a myriad of computational resources. From a software point of view, this support represents the bottommost layer of a software stack that is commonly used to describe the Grid in architectural terms. This architecture is depicted in Figure 2.2.

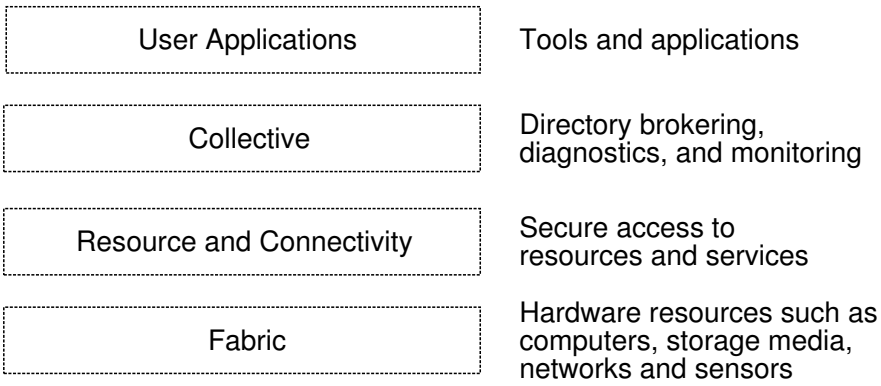


Figure 2.2: The Grid software stack (Foster and Kesselman, 2003a)

The stack is composed of four layers: *Fabric*, *Resource and Connectivity*, *Collective* and *User Applications*. Roughly, the Resource and Connectivity layer consists of a set of protocols capable of being implemented on top of many resource types (e.g. TCP, Hyper Text Transfer Protocol (HTTP)). Resource types are defined at the Fabric layer, which in turn are used to construct metaservices at the Collective layer, and Grid applications at the User Applications layer. The main characteristics of each layer are described next:

- *Fabric*: As mentioned above, this layer represents the physical infrastructure of the Grid, including resources such as computing nodes and clusters, storage systems, communication networks, database systems and sensors, which are accessed by means of Grid protocols. Generally speaking, resources at this layer are entities that may be either physical (e.g. disks, servers, PCs, etc.) or logical (e.g. a distributed file system, a computer cluster, etc.). Interaction with logical resources may involve internal, technology-specific protocols (e.g. Network File System (NFS) for distributed file access, middleware-specific job submission protocols, etc.), but these are out of the scope of the protocols considered by the Grid architecture as protocols for accessing Fabric resources.
- *Resource and Connectivity*: Defines protocols to handle all Grid specific transactions between different resources on the Grid. Protocols at this layer are further categorized as connectivity-related protocols, which enable the secure exchange of data between Fabric layer resources and perform user authentication, and resource-related protocols, which permit authenticated users to securely negotiate access to, interact with, control and monitor Fabric layer resources:
  - Connectivity-related protocols: Enable the exchange of data between Fabric layer resources. On top of the core communication services (transport, routing and naming), authentication protocols are used for verifying the identity of both users and resources. At present, communication protocols are mainly derived from the TCP/IP protocol stack. Similarly, many security mechanisms originally designed for the Internet have been redefined/extended to be used in Grid settings. However, in the near future, the changing nature of the Grid could demand the creation of a new brand of protocols that take into account special communication and security requirements.
  - Resource-related protocols: Having established identity, Grid users needs to interact with resources and services. These protocols build on communication and authentication protocols and define mechanisms to operate on individual resources. There are two classes of resource-related protocols: *information protocols*, which are used to gather information about the anatomy and state of a resource (e.g. processing power and current average load of a computing node), and *management protocols*, which are used to negotiate access to a shared resource. Management protocols usually prescribe mechanisms for specifying resource requirements (e.g. desired quality of service) and the operations that can be performed on the resource (e.g. remote job submission, disk space advanced reservation).

- *Collective*: The collective layer contains protocols and services associated with capturing interactions across collections of resources. Functionality offered at this layer include:

  - Directory services, which allow users to discover resources by their attributes (e.g. type, availability, size, load, etc.). An example of such a kind of service is MDS-2 (Czajkowski et al., 2001), an information and discovery service for the Grid which is part of the Globus Toolkit.
  - Coallocation, scheduling and brokering services, which allow users to request the allocation of one or more resources and the scheduling of tasks on suitable resources. Examples of these services are AppLeS (Berman et al., 2003) and Nimrod-G (Abramson et al., 2000).
  - Monitoring and diagnosis services, which detect and handle failures, overloads, security threats or attacks, and so on.
  - Data replication services, which are in charge of optimal management of storage (and to a lesser extent network and computing) resources to maximize data access performance with respect to various metrics (e.g. response time, throughput, reliability, etc.).

- *User Applications*: Each one of the previous layers expose well-defined protocols and APIs that provide access to services for resource management, data access, resource discovery and interaction, an so on. On the other hand, the User Application layer comprises the applications that operate within a VO, which are built upon Grid services by means of those APIs and protocols.

It is worth noting that some "applications" within the topmost layer may in turn be Grid programming facilities such as frameworks and middlewares, exposing themselves protocols and APIs upon which more complex applications (e.g. workflow systems) are created. In fact, these facilities can be seen as the "wall socket" by which applications are connected to the Grid. Application developers are likely to use high-level software tools that provide a convenient programming environment and isolate the complexities of the Grid, rather than use Grid services directly.

However, applications that have not been written to run on the Grid still have to be adapted in order to use the functionality provided by Grid programming facilities. In other words, these kind of applications need to be gridified so they can take advantage of Grid services and resources through a specific middleware or framework. As a consequence, an extra development effort is required from application programmers, which might not have the necessary skills or expertise to port their applications to the Grid. To sum up, the foreseen goal of gridification is to let conventional applications benefit from Grid services without requiring these applications to be modified. The concept of gridification is discussed in Section 2.3.

The next section takes a closer look at Grid services focusing on its supporting technologies.

2.2 Service-Oriented Grids

One of the most important technological changes that have occurred since the Grid Computing paradigm was proposed is the wide acceptance of Web service technologies (Stockinger, 2007). Basically, a Web Service is a piece of functionality with a well-defined interface that can be located and accessed by means of conventional Web protocols (Vaughan-Nichols, 2002; Curbera et al., 2003; Martin, 2001). To date, Web Services have been successfully employed in contexts such as Business-to-Business (B2B) and e-commerce applications. For example, many popular Web sites such as Amazon[4], eBay[5] and Google[6] offer Web Services for applications that expose the same information and functionality a user can access by using a regular Web browser (e.g. for performing a Google search, or buying a book in Amazon or eBay).

In the Grid arena, Web Services standards such as Service-Oriented Architecture Protocol (SOAP) (W3C Consortium, 2007a), Web Service Definition Language (WSDL) (W3C Consortium, 2007b) and Universal Description, Discovery, and Integration (UDDI) (OASIS Consortium, 2004) have gained great popularity. Today, these standards play a fundamental role, since they greatly help in providing a satisfactory solution to the problem of heterogeneous systems integration, thus supplying the basis for future Grid technology which will clearly need to be highly interoperable to meet the needs of global enterprises. In fact, the major Grid standardization efforts such as OGSA (OGSA-WG, 2005) and the WSRF (OASIS Consortium, 2006) heavily rely on Web Services technologies. In addition, Grid middlewares are evolving from their pre-Web Service state to new versions based on Web Services (Atkinson et al., 2005). For instance, research has been being done to integrate Condor with Web Services (Chapman et al., 2004; Chapman et al., 2005), and Globus has recently embraced WSRF. In summary, the current trend is to see the Grid as a provider of services –materialized as Web Services– on top of which complex Grid applications can be constructed by invoking, composing and orchestrating these services.

2.2.1 Web Services standards

Web Services have proven to be a suitable model to allow systematic interactions of distributed applications and integration of legacy platforms and environments. The Web Services model mostly relies on technologies based on Extended Markup Language (XML) (Bray et al., 2006), a structured language that extends and formalizes Hyper Text Markup Language (HTML). In this sense, the World Wide Web (W3C) Consortium has developed SOAP, a communication protocol entirely based on XML. Nowadays, SOAP is widely used and is included in most of the communication infrastructure proposed for integrating applications and Web Services. In addition, languages for describing Web Services have been developed. The most notorious example is WSDL, an XML-based language which allows developers to create service descriptions

[4]Amazon Bookstore: http://www.amazon.com/gp/aws/landing.html
[5]eBay: http://developer.ebay.com/DevProgram/index.asp
[6]Google Search Engine: http://code.google.com/apis/

as a set of operations over XML messages. From a WSDL specification, a program can discover the specific services a Web site provides, and how to use and invoke these services.

As a complement to WSDL, UDDI has been proposed. UDDI provides mechanisms for searching and publishing services written preferably in WSDL. UDDI is in essence a "store window" for Web Services consumers: Web Service providers such as enterprises or organizations register information about the services they offer, thus making this information available to potential clients. The information stored into UDDI registries ranges from files describing services to useful data (e-mail, Web pages, etc.) for contacting the associated provider.

The Web Services model is shown in Figure 2.3. Here, a Web Service is defined as an interface describing a collection of operations that are network-accessible through standardized XML messaging. WSDL is used to describe the software interface to the Web Service, and all interactions between any pair of components are supported through SOAP.

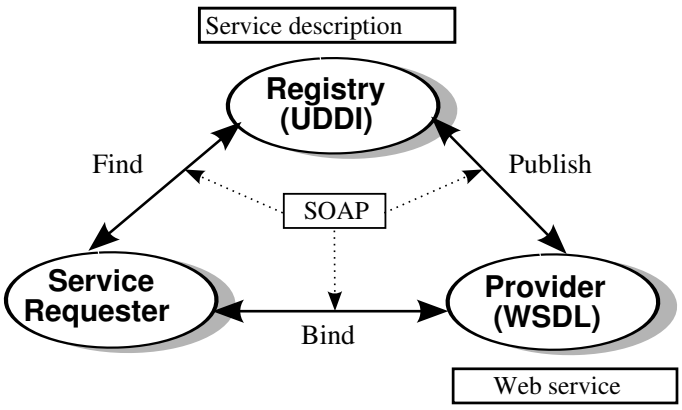


Figure 2.3: The Web Services model (Kreger, 2001)

The model encompasses three classes of elements: service providers, service requesters and service registries. A service provider creates a WSDL document describing its Web Service and publishes this document to a service registry such as UDDI. A service requester can use a registry to find a Web Service that matches its needs and retrieve the corresponding WSDL document. Using the information provided by a WSDL document, a service requester invokes the operations of the provider's Web Service.

SOAP, WSDL and UDDI are described next.

2.2.1.1 SOAP

SOAP is a communication protocol for exchanging XML-based messages over computer networks, normally using HTTP or Secure HTTP (HTTPS). As explained, SOAP represents the

foundation layer of the Web Services model, providing a basic messaging framework on which clients and services can build on.

Although there are several types of messaging patterns in SOAP, the most common is Remote Procedure Call (RPC), in which a *client* sends a request message to a *server* residing at a remote location. Then, the server processes the message and sends a response message back to the client. The RPC pattern is certainly not new, as it has been already implemented in distributed technologies such as Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA).

Since based on XML, SOAP messages can be read by humans, thus potentially simplifying debugging. However, the verbose nature of XML can cause the protocol to render slow message processing times and excessive usage of network bandwidth with respect to alternative representations for messages, such as binary formats. In any case, SOAP is versatile enough to leverage ubiquitous transport protocols like TCP, File Transfer Protocol (FTP) and Simple Mail Transfer Protocol (SMTP). Besides, using SOAP over HTTP allows for easier communication between hosts that have connectivity restrictions (e.g. hosts behind proxies or firewalls) than previous RPC-like technologies.

2.2.1.2 WSDL

WSDL is a standard for describing services and software components in a manner independent of any particular programming language. A WSDL Web Service definition is a XML document comprising a *service description*, which define the service interface, and *implementation details*, which specifies how the interface maps to concrete communication protocols and endpoint addresses. By establishing this separation, WSDL allows to multiple *bindings* for the same service interface. For example, a single service implementation might support bindings based on one or more distributed communication protocols, and an locally optimized binding (e.g. inter-process communication) for interactions between clients and services of the same host.

Figure 2.4 (a) illustrates the service description corresponding to a Web Service for retrieving files (StorageService), whereas Figure 2.4 (b) shows its implementation details. The <portType> element abstractly defines the Storage-Service interface by specifying a “getFile” operation. Operations are specified by an <operation> element, which defines the messages used to implement a specific interaction pattern (e.g. request-response, asynchronous, etc.) for the operation. In our example, the “getFile” operation has an input message (getFileRequest) and output message (getFileResponse), thus interaction between clients and service is synchronous.

The <message> element defines the messages used in either direction to implement an operation. A message is composed of zero or more parts, which have an associated datatype. Both input and output messages in our example define messages that are composed of a single string part.

```
<wsdl:definitions xmlns:tns="..." targetNamespace="...">
  <!-- Custom datatypes definitions -->
  . . .
  <message name="getFileInfoRequest">
    <part name="filename" type="xsd:string">
  </message>

  <message name="getFileInfoResponse">
    <part name="info" type="xsd:string">
  </message>

  <portType name="StorageServicePort">
    <operation name="getFileInfo">
      <input message="getFileInfoRequest">
      <output message="getFileInfoResponse">
    </operation>
  </portType>
  . . .
```

(a) Service description

```
  . . .
  <binding name="StorageServiceSoapBinding"
           type="StorageServicePort">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation>
      <soap:operation
        soapAction="http://example.com/getFileInfo"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
</wsdl:definitions>
```

(b) Implementation details

Figure 2.4: Example showing the elements of a WSDL definition

WSDL provides a set of built-in primitive datatypes, and also offers an extension mechanism to create custom datatypes. From an object-oriented perspective, port types, operations and messages can be thought as class interfaces, method and arguments, respectively.

Finally, binding details are specified within a <binding> element (see Figure 2.4 (b)), which specifies the messaging and transport protocol, messaging style (document or RPC) and data--encoding model (literal or SOAP-based) used to communicate messages. As the reader can see from the figure, the "getFile" operation uses SOAP messaging over HTTP transport, document messaging style and literal encoding.

2.2.1.3 UDDI

The UDDI specification is another important member of the group of related standards for Web Services. UDDI defines methods for publishing and discovering Web-accessible services. Central to UDDI is the concept of *registry*, which is a collection of one or more servers (or *nodes*) that support the UDDI specification. Unlike registries, nodes usually perform a reduced set of the functionality defined in the specification.

The purpose of UDDI registries is to maintain metadata about individual businesses and theirs corresponding services. UDDI provide mechanisms to classify and catalog this information to allow Web Services to be discovered and consumed by user applications and other services. Roughly, a UDDI registry consists of the following components:

- "White pages", which contain location and contact information about individual service providers (e.g. address, zip code, email, Web page, etc.),
- "Yellow Pages", which offer categorized views of published Web Services based on standard taxonomies and classification systems such as North American Industry Classification

System (NAICS)[7] or Universal Standard Products and Services Codes (UNSPSC)[8], and

- "Green Pages", which provide technical information about services exposed by providers, such as security and transport protocols supported by a given Web Service, parameters necessary to invoke or access the service, etc.

UDDI registries are accessed by client applications through the UDDI inquiry API, a standard set of functions for inspecting registry entries which are currently available for a variety of programming languages. In order to browse and query a registry, applications must firstly obtain an *authentication token* for the registry. Authentication tokens protect the information stored in UDDI registries by preventing malicious or unauthorized access.

2.2.2 WSRF

A major limitation of the Web Services model is that services are by itself *stateless*. In other words, the model does not explicitly prescribe mechanisms to record the "conversation" between a service requester and a Web Service. For example, an online airline reservation system that provides Web Services for flight booking must maintain state information about flight status and reservations made by specific customers. This fact clearly limits the potential of the model, and puts an extra burden on developers, since both applications and services are responsible for managing state information across invocations.

WSRF is a family of specifications for implementing stateful Web Services. WSRF provides a set of standard operations that Web Services may implement to become stateful. These operations allow individual services to be associated to one or more *resources*, whose solely purpose is to store the state information corresponding to a single invocation from a specific requester to a service. Each resource have a unique identifier (usually an Uniform Resource Identifier (URI)). Whenever a client application wants to perform a stateful interaction with a Web Service, the service is instructed to use a particular resource. Besides the notion of explicit resource referencing, WSRF includes a standardized set of operations to query and modify resource attributes. This support is mainly used by management and debugging tools to visualize and to track the state of the resources associated to a Web Service.

Figure 2.5 illustrates the components that are involved in the interaction of a service requester and a WSRF-enabled Web Service (labeled (1) in the figure). A Web Service is basically a piece of functionality whose implementation is deployed within a runtime environment (2), such as a Web server or a J2EE-compliant application server. This runtime environment hosts the implementation code of the Web Service and dispatches all invocations to the service. The

[7]NAICS: www.census.gov/naics
[8]UNSPSC: http://www.unspsc.org

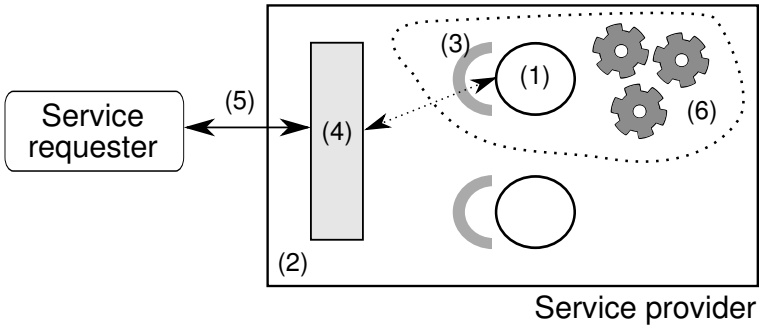


Figure 2.5: WSRF-enabled Web Services: operational components

Web Service's interface (3) describes –usually by means of WSDL– the service's functionality in terms of a set of operation that may be invoked by service requesters. The runtime environment also provides a message processing facility (4) that can handle the communication between the Web service and the requester (5). Most modern message processing facilities for Web Services support the SOAP protocol. Finally, the Web Service implementation is responsible for receiving a message, and processing the message, potentially interacting with stateful resources (6) and other Web Services. Web Services may also play the role of a service requester, initiating conversations with other Web Services.

2.2.2.1 WSRF Core Specifications

WSRF consists of a collection of four different specifications, which relate to a greater or lesser extent to the management of WSRF-like resources:

- *WS-ResourceProperties*: As mentioned above, a resource is composed of zero or more properties. For example, a resource representing a flight reservation may have properties like date, origin, destination and seat number. WS-ResourceProperties specifies how resource properties are defined and accessed.

- *WS-ResourceLifetime*: In general, resources have non-trivial lifecycles. Namely, resources can be created and destroyed at any time. The WS-ResourceLifetime provides basic mechanisms to manage the lifecycle of resources.

- *WS-ServiceGroup*: A ServiceGroup is a heterogeneous collection of Web Services. ServiceGroups can be used to form a variety of collections of services or resources, including registries of Web Services and their associated resources. The WS-ServiceGroup specification standardizes the way Service-Groups are managed, providing operations related to group management (e.g. adding/removing members) and basic querying (e.g. find a service that meets condition X).

- *WS-BaseFaults*: This specification simply aims at providing a standard, implementation-neutral way of reporting faults produced during the invocation of a Web Service.

These specifications are commonly used in conjunction with other Web Service specifications, such as WS-Notification and WS-Addressing[9]. WS-Notification allows a Web Service to act as a notification producer, and clients to be notification subscribers. As a consequence, subscribers are notified whenever a change occurs in the Web Service (e.g. a property in a resource have been modified). On the other hand, WS-Addressing provides mechanisms to easily reference Web Services and its associated resources.

2.2.3 OGSA

OGSA is the most popular reference architecture for implementing Grid systems, which is fully based on the principles of Service-Oriented Architecture (SOA) (Huhns and Singh, 2005). The basic building brick of SOAs is the notion of *service*, that is, an entity that provides special capabilities to its clients by exchanging messages. A service is defined by an *interface* that contains one or more *operations*. This interface represents the "contract" to which clients agree in order to interact with the service. Therefore, by clearly defining service interfaces, a great degree of flexibility is achieved, specially on how services are implemented or where they are located. As the reader can see, these principles also apply smoothly to the Web Services model, in which standards are provided to specify both service interfaces (i.e. WSDL) and messaging (i.e. SOAP), thus achieving interoperability. In essence, the aim of OGSA is to leverage the benefits of both SOA and Web Services standards to simplify the development of highly decoupled, interoperable, reusable Grid applications.

By hiding functionality behind a common message-oriented service interface, SOAs promotes *service virtualization* (Foster et al., 2003), which is a fundamental requirement of Grid systems. For example, a storage service might expose a "transfer file" operation to transfer data from one Grid node to another. In this way, a user should be able to invoke this operation on a particular instance of the storage service without knowing nothing about how that instance implements the transfer file operation (e.g. the underlying transport protocol for transferring the file).

A simplified view of OGSA is shown in Figure 2.6. As depicted, OGSA is built on top of Web Services standards, which provide the basis for interoperability and platform-independence across a variety of different Grid environments. On top of those standards, WS-* specifications address important issues related to basic service semantics such as service lifetime, fault management, notification mechanisms, service identification, and so on. In this context, a Web Service that adheres to WS-* standards is known as a *Grid Service*. Note that these elements can be viewed as a reference materialization of the lowest layers of the Grid architecture presented in past sections, in the sense that Grid services can be used to perform basic virtualization of Grid resources.

[9]Hence the name of "WS-* standards" given to these family of specifications

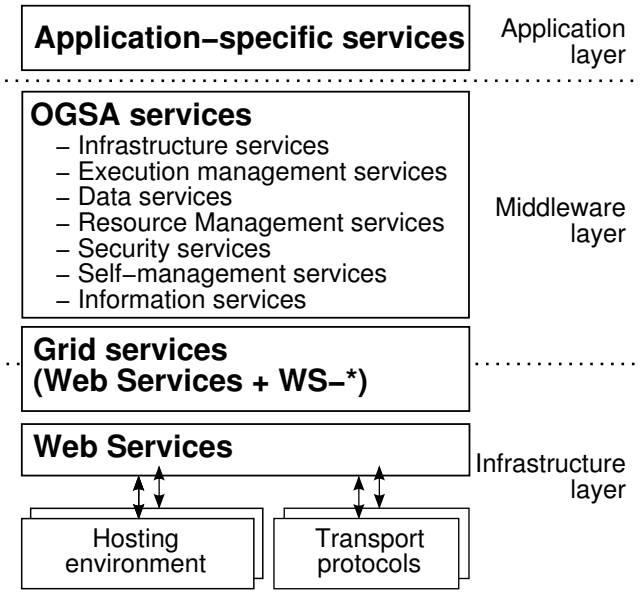


Figure 2.6: The core elements of OGSA

Grid services serve as the basement for the *OGSA services*. OGSA services are basically specialized Grid services that materialize behavior commonly found when building large-scale distributed systems. Examples are service discovery, authentication, quality of service and policy agreement, resource monitoring, data integration and management, security, to name a few. OGSA services can be seen as "metaservices" that materialize the middle layers of the Grid architecture. Finally, on top of OGSA services, more complex, domain-specific services may be built.

An interesting fact about the OGSA reference architecture is the way it has influenced the development and evolution of service-oriented Grid technologies and standards. Almost all specifications related to Web and Grid Services are in a constant process of revision and refinement. In addition, the Grid research community is progressively acquiring better understanding on what services a Grid should provide, and how these services should be materialized according to current Grid standards. This translates into more and more sophisticated, reusable and interoperable middleware Grid services for Grid application developers.

As a complement, many researchers have been also looking for pushing Grid technologies further by following the opposite direction, namely providing toolkits, frameworks and programming facilities to simplify the consumption of services from within user applications. The ultimate goal of these technologies is to produce effective techniques and methods to allow application developers to benefit from Grid resources without the need for either knowing details of the Grid or modifying applications. Certainly, gridifying applications is a difficult and challenging problem that has motivated a significant amount of research over the last years. In this sense, the next section provides a brief overview of Grid technologies focused on providing solutions to this problem.

2.3 Gridification Technologies: Origins and Evolution

It is difficult to determine exactly when the term "gridification" was first introduced, but the idea of achieving easy pluggability of ordinary applications into the Grid surely took a great impulse at the time the analogy between electrical power grids and computational Grids was established. Nowadays, the concept of gridification is widely recognized among the Grid research community, and many researchers explicitly use the term "gridification" to refer to this idea (Ho et al., 2003; Wang et al., 2004; Imade et al., 2004; Ho et al., 2006; Taffoni et al., 2007; Mateos et al., 2008; Mateos et al., 2007a). The evolution of Grid technologies from the point of view of gridification is presented in the next paragraphs.

The first attempts to achieve gridification began with the use of popular technologies traditionally employed in the area of Parallel and Distributed Computing such as Parallel Virtual Machine (PVM) (Geist et al., 1994), Message Passing Interface (MPI) (Dongarra and Walker, 1996) and RMI (Downing, 1998). Basically, the underlying programming models of these technologies were reconsidered to be used in Grid settings, yielding as a result standardized Grid programming APIs such as MPICH-G2 (Karonis et al., 2003) (message passing) and GridRPC (Nakada et al., 2005) (remote procedure call). Grid applications developed under these models are usually structured as "masters" and "workers" components communicating through ad-hoc protocols and interaction mechanisms. For example, programmers must explicitly supply API code to either synchronously or asynchronously send/perform messages/calls within these components. Developers are also responsible for managing parallelization and location of application components. Specifically, aspects such as threading and resource management (e.g. physical information about the nodes involved in a computation) are entirely handled by programmers. As a consequence, at this stage there is not a clear idea of Grid resource *virtualization* yet. Consequently, gridification was mainly concerned with manually taking advantage of the Grid infrastructure, that is, the Fabric layer of the software stack in Figure 2.2.

The second phase of the evolution of gridification technologies involved the introduction of Grid middlewares. Some of them were initially focused on providing services for automating the scavenging of processing power, memory and storage resources (e.g. Condor, Legion), while others aimed at raising the level of abstraction of Grid functionality by providing *metaservices* (brokering, security, monitoring, etc.). A representative example of a middleware in this category is Globus, which have become the *de facto* standard for building Grid applications. Overall, users are now supplied with a concrete virtualization layer that isolates the complexities of the Grid by means of services. In fact, technologies like MPICH-G2 and GridRPC are now seen as middleware-level services for communication rather than Grid programming facilities *per se*. Gridification is therefore conceived as the process of writing/modifying an application to utilize the various services provided by a specific Grid middleware. As the reader can observe, the main goal of gridification technologies at this stage is to materialize the middle layers of the Grid software stack.

The step that followed the appearance of the first Grid middlewares was the introduction of

Grid programming toolkits and frameworks. In this step, the problem of writing applications for the Grid received more attention and the community recognized common behavior shared by different Grid applications. The idea behind these technologies is to provide generic APIs and programming templates to unburden developers of the necessity to know the many particularities for contacting individual Grid services (e.g. protocols and endpoints), to capture common patterns of service composition (e.g. secure data transfer), and to offer convenient programming abstractions (e.g. master-worker templates). The most important contribution of these solutions is to capture common Grid-dependent code and design in an application-independent manner. These tools can be seen as an incomplete application implementing non-application specific functionality, with *hot-spots* or *slots* where programmers should put application specific functionality in order to build complete applications (Johnson, 1997; Codenie et al., 1997).

For example, the Java CoG Kit (Globus Alliance, 2006) provides an object-oriented, framework-based interface to Globus-specific services. The Grid Application Toolkit Grid Application Toolkit (GAT) (Allen et al., 2003; Allen et al., 2005) and SAGA (Goodale et al., 2006) are similar to the Java CoG Kit but they offer APIs for using Grid services that are independent of the underlying Grid middleware. With respect to template-based Grid frameworks, some examples are MW (Goux et al., 2000), AppLeS Master-Worker Application Template (AMWAT) (Berman et al., 2003) and JaSkel (Ferreira et al., 2006). All in all, the goal of these tools is to make Grid programming easier. The conception of gridification at this phase does not change too much from that of the previous one, but Grid programming is certainly done at a higher level of abstraction. As a consequence, less design, code, effort and time is required when using these tools.

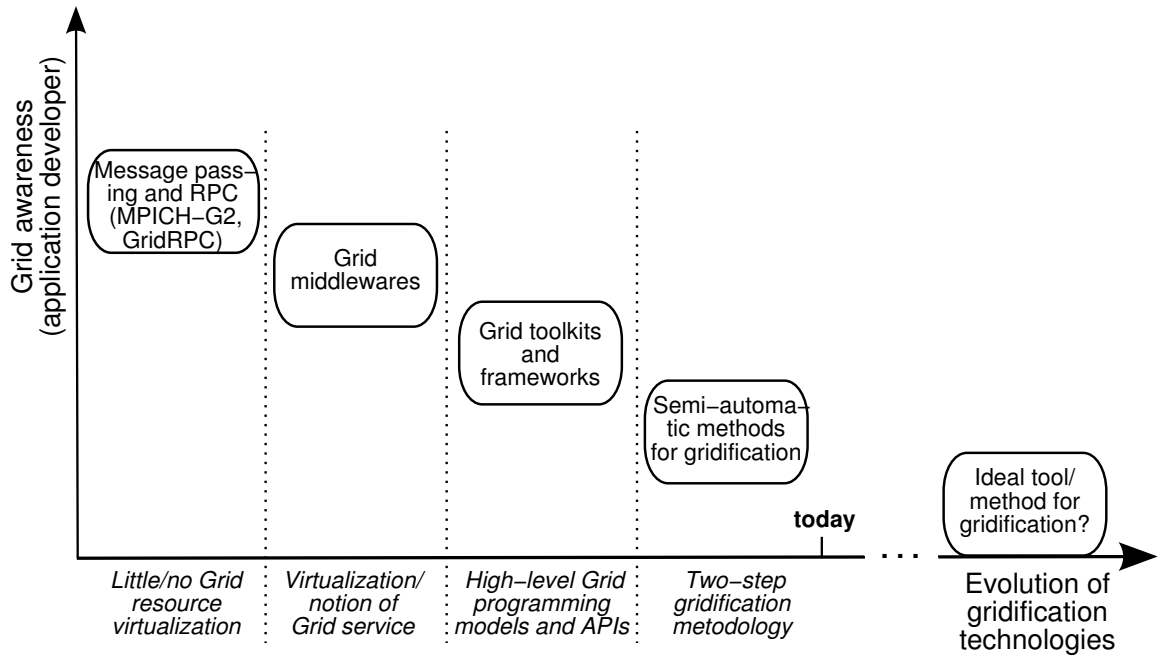


Figure 2.7: Origins and evolution of gridification technologies

Up to this point, the most remarkable characteristic shared among the above technologies is that gridification is done in a *one-step* process, that is, there is not a clear separation between the tasks of writing the pure functional code of an application and adding it Grid concerns. The Grid technology being used plays a central role during the entire Grid application development process, since developers Grid-enable applications as they code them by keeping in mind a specific Grid middleware, toolkit or framework. Therefore, technologies promoting *one-step* gridification assume developers have a solid knowledge on Grid programming and runtime facilities.

Alternatively, there are a number of Grid projects promoting what it might be called a *two-step* gridification methodology, which is intended to support users having little or even no background on Grid technologies. Basically, the ultimate goal of this line of research is to come out with methods that let developers to focus first on implementing and testing the pure functional code of their applications, and *then* to semi-automatically Grid-enable them. Note that, besides having potential benefits in terms of logic code stability, this methodology is suited for gridifying applications that were not initially designed nor thought to run on the Grid. It is worth noting that technologies under this gridification paradigm can be seen a complement to the ones previously described. In fact, active research is being done to develop more usable and intuitive Grid programming models, toolkits and middlewares.

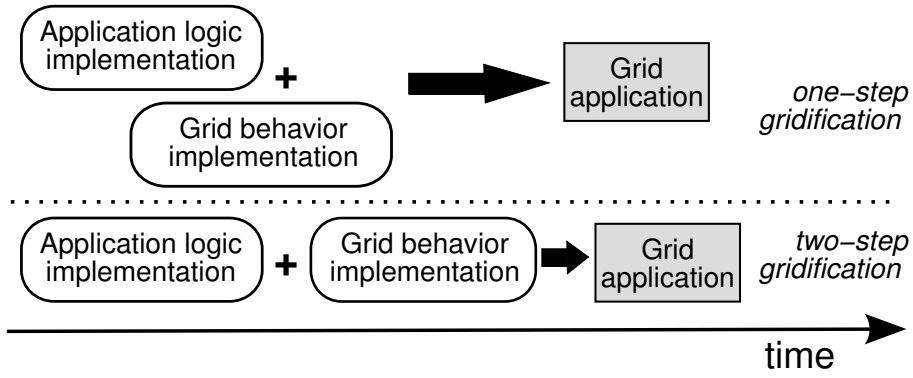


Figure 2.8: One-step and two-step gridification in a nutshell

One-step and two-step gridification methodologies are illustrated in Figure 2.8 (above and below the dashed line, respectively). One-step gridification suggests an overlapping between the tasks of writing the application logic and supplying code for using Grid technologies. In contrast, two-step gridification is based on making these tasks to be carried out sequentially.

Figure 2.7 shows how the evolution of Grid technologies have reduced the knowledge that is necessary to gridify an application. As depicted in the figure, four separate phases in this evolution can be identified. Transitions between two consecutive phases is given by a radical

change in the conception of the notion of gridification. At the first phase, "gridify" means to manually use the Grid infrastructure. At the second phase, virtualization of Grid resources through services is introduced; "gridify" refers to adapt applications to use Grid services. The third phase witnessed the introduction of the first Grid development technologies materializing common behavior of Grid applications, therefore gridification takes place at a higher level of abstraction. Finally, the fourth phase incorporated the notion of two-step gridification: Grid technologies recognized the need to provide methods to *transform* ordinary applications to Grid-aware ones with little effort.

Certainly, the relation between the two axis is not linear, but it is descriptive enough to get an idea about the consequences of gridification in the long term. As illustrated, the *ideal* method for gridification would yield an hypothetical value for Grid awareness equals to *zero*, that is, the situation in which developers can effectively exploit the Grid without explicitly using any Grid technology in their code.

2.4 Conclusions

Grid Computing represents the next logical step in distributed computing. Although the concept is fairly new, Grid standards and technologies have developed rapidly, and the paradigm is steadily moving from the academia towards commercial adoption. The major purpose of a Grid is to virtualize distributed resources such as processing power, storage systems, networking facilities, and so on, to solve very complex problems in science, engineering and commerce.

As Grids evolve, some trends become evident, and the most remarkable is the convergence of Grid technologies and SOAs. The envisioned result of this combination is a powerful computing infrastructure offering specialized, interoperable and reusable services to user applications. However, the extremely high complexity inherent to this service infrastructure calls for not just new Grid development tools but for novel methods to transparently consume Grid services from within conventional applications.

Surprisingly, even though the Grid research community has been well aware of the existence of the gridification problem from several years now, few publications are available on the subject and, in those cases, only a small part of the problem is explored and analyzed (Mateos et al., 2007a). Specifically, previous works towards better characterizing Grid technologies and their relationship with the gridification problem are:

- (Bal et al., 2003), in which the authors point out the programming and deploying complexity inherent to Grid Computing. They state there is a need for tools to allow application developers to easily write and run Grid-enabled applications, and also identify a taxonomy of Grid application-level tools that is representative enough for many projects in the Grid community. This taxonomy distinguishes between two classes of application-level tools for the Grid: programming models (i.e. tools that build on the Grid infrastructure and provide high-level programming abstractions) and execution environments (i.e. software tools

into which users deploy their applications). However, the discussion is clearly focused on illustrating how these models and environments can be used to develop Grid applications from scratch, rather than gridify existing applications.

- (Kielmann et al., 2006), in which several functional and non-functional properties that a Grid programming environment should have are identified, and some tools based on these properties are reviewed. The survey concludes by deriving a generic architecture for building programming tools that are capable of addressing the whole set of properties, which prescribes a component-based approach for materializing both the runtime environment and the application layer of a Grid platform. As in the previous case, the work does not discuss aspects related to gridification of existing applications either.

- A survey on technologies for wide-area distributed computing can be found in (Baker et al., 2002), where the most predominant trends for accelerating Grid application programming and deployment are identified. This work aims at providing an exhaustive list of Grid Computing projects ranging from programming models and middlewares to application--driven efforts, while our focus is exclusively on methods seeking to attain easy pluggability of conventional applications into the Grid. A similar work is (Venugopal et al., 2006), in which a thorough examination of technologies for the materialization of Data Grids –those providing services and infrastructures to manage huge amounts of data– is presented.

The next chapter presents a taxonomic framework to help in gaining an insight on the various dimensions of the gridification problem, and surveys the most relevant projects that attempt to address this problem. This thesis provides an alternative approach for gridifying applications that is based on two-step gridification. Consequently, the discussion will be strictly circumscribed to those gridification methods that are aimed at producing Grid-aware applications by taking as input existing ordinary applications.

Chapter 3

Related Work

The purpose of this chapter is to summarize the state of the art on Grid development approaches focusing specifically on those that target easy *gridification*, that is, the process of adapting an ordinary application to run on the Grid. It is worth mentioning that the discussion does not exhaustively analyze the current technologies for implementing or deploying Grid applications. Instead, this chapter examines existing techniques to gridify software that has not been at first thought to be deployed on Grid settings, such as desktop applications or legacy code. In order to limit the scope of the analysis, the discussion will focus our discussion on the amount of effort each proposed approach demands from the user in terms of source code refactoring and modification. As a complement, for each approach, the chapter will analyze the anatomy of applications after gridification and the kind of Grid resources they are capable of transparently leverage.

The rest of the chapter is organized as follows. The next section surveys some of the most representative approaches for gridifying applications. Later, Section 3.2 summarizes the main features of the surveyed approaches, and presents several taxonomies to capture the big picture of the subject. Based on these taxonomies, Section 3.3 identifies common characteristics and trends. Finally, Section 3.4 concludes by stating common problems among the proposed approaches, which represent the motivation of the present thesis.

3.1 Gridification Projects

In light of the gridification problem, a number of studies have proposed solutions to port existing software to the Grid. For example, (Ho et al., 2003) presents an approach to assist users in gridifying complex engineering design problems, such as aerodynamic wing design. Similarly, (Wang et al., 2004) introduces a scheme of gridification specially tailored to gridify scientific legacy code. In addition, (Kolano, 2003) proposes an OGSA-compliant *naturalization*[1] *service*

[1]The American Heritage Dictionary defines naturalization as "adapting or acclimating (a plant or an animal) to a new environment; introducing and establishing as if native".

for the Globus platform that automatically detects and resolves software dependencies (e.g. programs, system libraries, Java classes, among others) when running CPU-intensive jobs on the Grid.

Although the above technologies explicitly address the problem of achieving easy gridification, they belong to what it might be identified as early efforts in the development of true gridification methods, which are characterized by solutions lacking generality and targeting a particular application type or domain. Nonetheless, there are a number of projects attempting to provide more generic, semi-automatic methods to gridify a broader range of Grid applications, mostly in the form of sophisticated programming and runtime environments. In this sense, Sections 3.1.1 to 3.1.10 present some of these projects.

3.1.1 GEMLCA

Grid Execution Management for Legacy Code Architecture (GEMLCA) (Delaittre et al., 2005) is a general architecture for transforming legacy applications to Grid services without the need for code modification. GEMLCA let users to deploy a legacy program written in any programming language as an OGSA-compliant service. The access point for a client to GEMLCA is a front-end offering services for gridifying legacy applications, and also for invoking and checking the status of running Grid services. An interesting feature of this front-end is that it is fully integrated with the P-GRADE (Kacsuk and Sipos, 2005) workflow-oriented Grid portal, thus allowing the creation of complex workflows where tasks are actually gridified legacy applications.

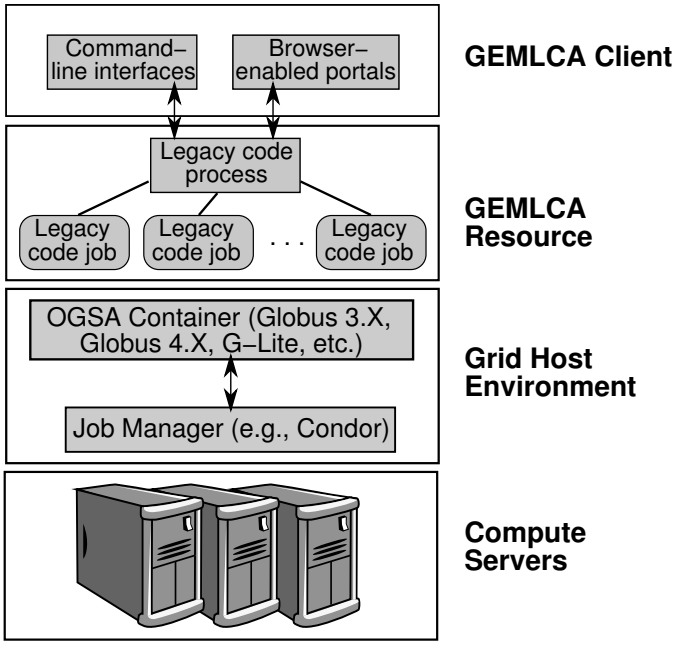


Figure 3.1: Overview of GEMLCA

GEMLCA aims at providing an infrastructure to deploy legacy applications as Grid services without reengineering their source code. As depicted in Figure 3.1, GEMLCA is composed of four layers:

- *Compute Server*s: Represents hardware resources such as PCs, servers and clusters on which legacy applications in the form of binary executables are potentially available. Basically, the goal of GEMLCA is to make these applications accessible through Web Services-enabled Grid services.
- *Grid Host Environment*: Implements a service-oriented Grid layer on top of a specific OGSA-compliant Grid middleware. Current distributions of GEMLCA supports Globus version 3.X and 4.X.
- *GEMLCA Resource*: Provides portal services for gridifying existing legacy applications.
- *GEMLCA Client*: This layer comprises the client-side software (i.e. command-line interfaces and browser-enabled portals) by which users may access GEMLCA services.

The gridification scheme of GEMLCA assumes that all legacy applications are binary executable code compiled for a particular target platform and running on a Compute Server. The Resource layer is responsible to hide the native nature of a legacy application by wrapping it with a Grid service, and processing service requests coming from users. It is up to the user, however, to describe the execution environment and the parameter information of the legacy application. This is done by configuring an XML-based file called Legacy Code Interface Description (LCID), which is used by the GEMLCA Resource layer to map Grid service requests to job submissions. LCID files provide metadata about the application, such as its executable binary path, the job manager and the minimum/maximum number of processors to be used, and parameter information, given by the name, type (input or output), order, regular expressions for input validation, and so forth. The following code presents the LCID file corresponding to the gridification of the Unix *mkdir* command:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE GLCEnvironment "gemlcaconfig.dtd">
<GLCEnvironment id="mkdir"
    executable="/bin/mkdir" jobManager="Condor"
    maximumJob="5" minimumProcessors="1">
  <Description>Unix mkdir command</Description>
  <GLCParameters>
    <Parameter name="-p" friendlyName="New folder"
        inputOutput="Input" order="0" mandatory="No">
      <initialValue/>
    </Parameter>
  </GLCParameters>
</GLCEnvironment>
```

As explained, the GEMLCA gridification process demands zero coding effort and little configuration from the user. In spite of this fact, users not having an in-depth knowledge about GEMLCA concepts may experience difficulties when manually specifying LCID files. In this sense, the GEMLCA front-end also provides user-friendly Web interfaces to easily describe and deploy legacy applications.

A more serious problem of GEMLCA is concerned with the anatomy of a gridified application. GEMLCA applications are essentially an ordinary executable file wrapped with an OGSA service interface. GEMLCA services serve request according to a very nongranular execution scheme (i.e. running the same binary code on one or more processors) but no internal changes are made in the wrapped applications. As a consequence, the parallelism cannot be controlled in a more grained manner. For many applications, this capability is crucial to achieve good performance.

3.1.2 GrADS

Grid Application Development Software (GrADS) (Vadhiyar and Dongarra, 2005) is a performance-oriented middleware whose goal is to optimize the execution of numerical applications written in C on distributed heterogeneous environments. GrADS puts a strong emphasis on application mobility and scheduling issues in order to optimize application performance and resource usage. Platform-level mobility in GrADS is performed through the so-called *Rescheduler*, which periodically evaluates the performance gains that potentially can be obtained by migrating applications to underloaded resources. This mechanisms is known as *opportunistic migration*.

Users wanting to execute an application contact the GrADS *Application Manager*. This, in turn, contacts the *Resource Selector*, which accesses the Globus MDS-2 service to obtain the available list of computing nodes and then uses the Network Weather Service (NWS) (Wolski et al., 1999) to obtain the runtime information (CPU load, free memory and disk space, etc.) from each of these nodes. This information, along with execution parameters and a user-generated execution model for the application, is passed forth to the *Performance Modeler*, which evaluates whether the discovered resources are enough to achieve good performance or not. If the evaluation yields a positive result, the *Application Launcher* starts the execution of the application using Globus job management services. Running jobs can be suspended or canceled at any time due to external events, such as user intervention.

GrADS provides a user-level C library called Stop Restart Software (SRS) that offers applications functionality for stopping at a certain point of their execution, restarting from a previous point of execution, and performing variable checkpointing. To SRS-enable an ordinary application, users have to manually insert instructions into the application source code in order to make calls to the SRS library functions. Unfortunately, SRS is implemented on top of MPI, so it can only be used in MPI-based applications. Nonetheless, as these applications are composed of a number of independent, mobile communicating components, they are more granular, thus

potentially achieving better use of distributed resources than conventional GrADS applications, that is, without using SRS.

3.1.3 GRASG

Gridifying and Running Applications on Service-oriented Grids (GRASG) (Ho et al., 2006) is a framework for gridifying applications as Web Services with relatively little effort. Also, in order to make better use of Grid resources, GRASG provides a scheduling mechanism that is able to schedule jobs accessible through Web Services protocols. Basically, GRASG provides services for job execution, monitoring and resource discovery that enhance those offered by Globus.

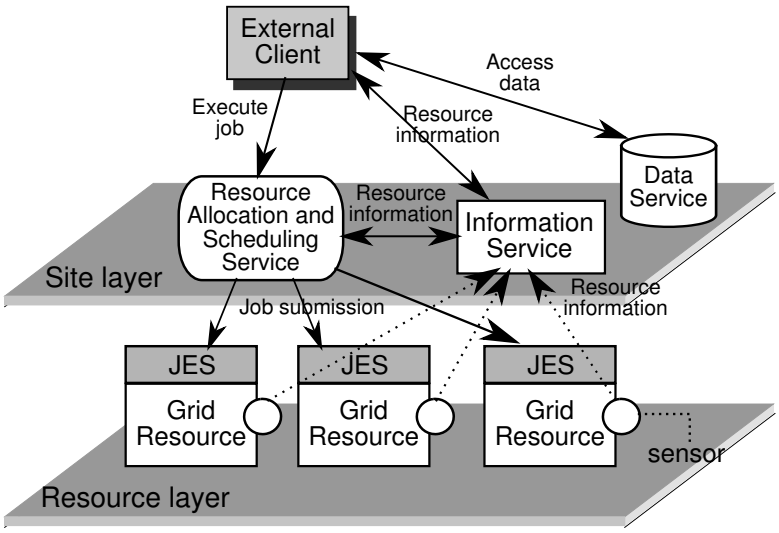


Figure 3.2: GRASG architecture

The architecture of GRASG is depicted in Figure 3.2. Its main components are four Web Services named Information Service (IS), Resource Allocation and Scheduling Service (RASS), Job Execution Service (JES), and Data Service (DS). Each Grid resource (i.e. a server) is equipped with the so-called *sensors* and wrapped with a JES. Sensors are responsible for capturing and publishing meta-information about their hosting resource (platform type, number of processors, installed applications, workload, etc.), while JES services are responsible for job execution and guaranteeing Quality of Service (QoS). More important, a JES wraps all the (gridified) applications installed on a server. External clients can execute gridified applications and "talk" to GRASG components by means of SOAP (W3C Consortium, 2007a), a well-known protocol for invoking Web Services.

The IS, RASS and DS services are placed on the *Site* layer, which sits on top of Grid resources. The IS periodically collects information about the underlying resources from their associated sensors, and use this information to satisfy resource requests originating either at the RASS or

an external client. The RASS bridges application clients to JESs. Specifically, the RASS is in charge of processing job execution requests coming from clients, allocating and reserving the needed Grid resources, monitoring the status of running jobs, and returning the results back to the clients. Lastly, the DS is used mainly for moving data among computation servers. It is implemented as a Web Service interface to GridFTP (Allcock et al., 2002), an FTP-based, high-performance, secure, reliable data transfer protocol for Grid environments.

GRASG conceives "gridification" as the process of deploying an existing application (binary executable) on a Grid resource. Once deployed, applications can be easily accessed through their corresponding JES, which stores all the necessary information (e.g. executable paths, system variables, etc.) to execute a gridified or a previously installed application. Like GEMLCA, application granularity after gridification is very coarse. To partially deal with potential performance issues caused by this problem, users can define custom scheduling and resource discovery mechanisms for a gridified application by writing new sensors that are based on shell or Perl scripts.

3.1.4 GridAspecting

GridAspecting (Maia et al., 2006) is a development process that is based on Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) to explicitly separate crosscutting Grid concerns in parallel Java applications. Its main goal is to offer guidelines for Grid application implementation focusing on separating the pure functional code as much as possible from the Grid-related code. Besides, GridAspecting relies on a subset of the Java thread model for application decomposition that enables for Grid application testing even outside a Grid setting.

GridAspecting uses a finer level of granularity for gridified components than GEMLCA and GRASG. GridAspecting assumes that ordinary applications can be decomposed into a number of independent *tasks*, which can be computed separately. As a first step, the programmer is responsible for identifying these tasks across the, yet non gridified, application code, and then encapsulate them as Java threads. Any form of data communication from the main application to its task threads should be implemented via parameter passing to the task constructor. As a second step, aspects have to be provided by the programmer in order to map the creation of a task to a job execution request onto a specific Grid middleware (e.g. Globus). At runtime, GridAspecting uses the AspectJ (Kiczales et al., 2001) AOP language to dynamically intercept all thread creation and initialization calls emitted by the gridified application, replacing them with calls to the underlying middleware-level execution services by means of those aspects.

Despite being relatively simple, the process requires the developer to follow a number of code conventions. However, applying GridAspecting results in a very modular and testable code. After passing through the gridification process, the functional code of an application is entirely separated from its Grid-related code. As a consequence, a different Grid API can be used without affecting the code corresponding to the application logic.

3.1.5 GriddLeS

Grid Enabling Legacy Software (GriddLeS) (Kommineni and Abramson, 2005) is a development environment that facilitates the construction of complex Grid applications from legacy software. Specifically, it provides a high-level tool for building Grid-aware workflows based on existing, unmodified applications, called *components*. Overall, GriddLeS goals are directed towards leveraging existing scientific and engineering legacy applications and easily wiring them together to construct new Grid applications.

The heart of GriddLeS is GridFiles, a flexible and extensible mechanism that allows workflow components to communicate between each other without the need for source code modification. Basically, GridFiles overloads the common file input/output primitives of conventional languages with functionality for supporting file-based interprocess communication over a Grid infrastructure. In this way, individual components behave as if they were executing in the same machine and using a conventional file system, while they actually interchange data across the Grid. It is important to note that GriddLeS is mainly suited for gridifying and composing legacy applications in which the computation time/communication time ratio is very high. Additionally, components should expose a clear interface in terms of required input and output files, so as to simplify the composition process and do not incur in component source code modification.

GridFiles makes use of a special language-dependent routine, called *FileMultiplexer*, which intercepts file operations and processes them according to a redirection scheme. Current materializations include local file system redirection, remote file system redirection based on GridFTP and remote process redirection based on sockets. When using process redirection, a multiplexer placed on the sending component is linked with a multiplexer on the receiving component through a buffered channel, which automatically handles data synchronization. In any case, the type of redirection is dynamically selected depending on whether the file identifier represents a local file, a remote file or a socket, and the target's location for the redirection (file or component) is obtained from the GriddLeS Name Server (GNS). The GridFiles mechanism is summarized in Figure 3.3.

The GriddLeS approach is simple yet very powerful. Applications programmers can write and test components without taking into account any Grid-related issues such as data exchanging, synchronization or fault-tolerance, which in turn are handled by the underlying multiplexer being used. Another interesting implication of this fact is that implemented components can transparently operate either as a desktop program or as a block of a bigger application. The weak point of GriddLes is that its runtime support suffers from portability problems, since it is necessary to have a new implementation for each programming language and operating system platform. Also, its implicit socket-based communication mechanism lacks the level of interoperability required by current Grids.

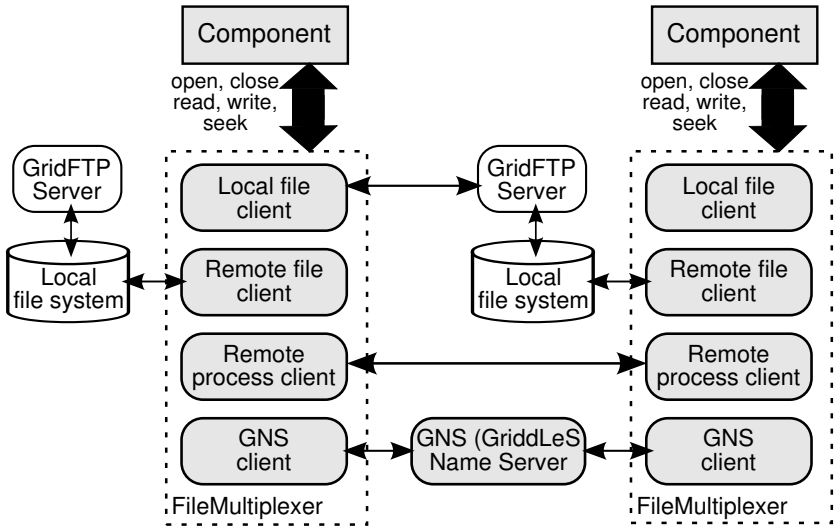


Figure 3.3: GridFiles: file request redirection

3.1.6 Ninf-G

Ninf-G (Takemiya et al., 2003) is a C/FORTRAN programming environment that aims at providing a simple Grid programming model mostly for non-computer scientists. It builds on top of the Globus toolkit and offers a reference implementation of the GridRPC specification. Ninf-G provides familiar RPC semantics so that the complicated structure of a Grid are hidden behind an RPC-like interface.

Figure 3.4 describes the architecture of Ninf-G, which is based on two major components: *Client Component* and *Remote Executable*. The Client Component consists of a client API and libraries for GridRPC invocations. The Remote Executable comprises a stub and system-supported wrapper functions, both similar to those provided by Java RMI or CORBA (Pope, 1998). The stub is automatically generated by Ninf-G from a special Interface Definition Language (IDL) file describing the interface of a remote executable. Both client and server programs are obtained after gridifying an application.

When executing a gridified application, the Client Component and the Remote Executable communicate with each other by using Globus services. First, the Client Component gets the IDL information for the server-side stub, comprising the remote executable path and parameter encoding/decoding information. This is done by means of MDS-2, the Globus network directory service. Then, the client passes the executable path to the Globus Grid Resource Allocation Manager (GRAM), which invokes the server-side part of the application. Upon execution, the stub requests the invocation arguments to the client, which are transferred using the Globus Input/Output service.

Roughly, the first step to gridify an application is to identify a client part and one or more server parts. The user should properly restructure its application whenever a server part cannot be

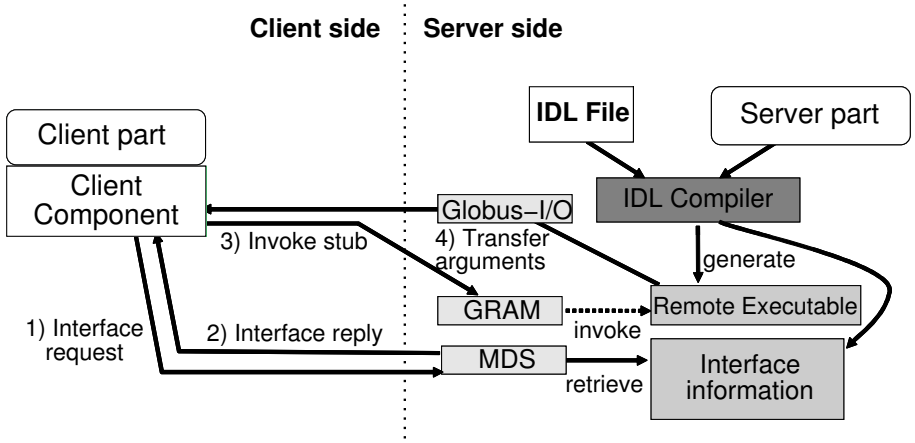


Figure 3.4: Ninf-G architecture

straightforwardly obtained from the code, such as merge the most resource-consuming functions into a new one and pick this latter as the server program. In any case, the user must carefully remove any data dependence between the client and the server program, or among server parts (e.g. global variables). Up to this point, the gridification process does not require to be performed within a Grid setting.

```
main(){
   pre_processing();
   call_library(args):
}
```

Gridification

```
main(){
   pre_processing();
   grpc_call(handle,
             "call_library",
             args);
}
```

```
main(){
   for (i=0;i<task_no;i++)
      task_processing(args);
}
```

Gridification

```
main(){
   for (i=0;i<task_no;i++)
      grpc_call_async(dest[i],
                      "task_processing",
                      args);
   grpc_wait_all(dest);
}
```

Figure 3.5: Gridifying applications with Ninf-G: typical scenarios

The next step is concerned with inserting Ninf-G functions into the client program so as to enable it to interact, via RPC, with its server parts(s). Ninf-G has a number of built-in functions for initiating and terminating RPC sessions and, of course, performing asynchronous or synchronous RPC calls. Typical scenarios when gridifying code with Ninf-G are illustrated in Figure 3.5.

Deploying a gridified application involves creating the executables on each server. First, the user must specify the interface for the server program(s) using Ninf-G IDL, which are used to automatically generate server-side stubs. Finally, the user must manually register this information in MDS-2. Although simple, these tasks can be tedious if several applications are to be gridified.

3.1.7 PAGIS

PAGIS (Webb and Wendelborn, 2003) is a Grid programming framework and execution environment suitable for unskilled Grid developers. PAGIS provides a component-based programming model that emphasizes on separating *what* an application does from *how* it does it. Roughly, putting an application to work onto the Grid with PAGIS first requires to divide the application into communicating components, and then to implement how these components are executed and controlled within a Grid environment.

A PAGIS application comprises a number of components connected through a network of unidirectional links called a *process network*. In PAGIS terminology, components and links are known as *processes* and *channels*, respectively. A process is a sequential Java program that incrementally reads data from its incoming channels in a first-in first-out fashion, transforms data, and produces output to some or all of its outcoming channels. At runtime, PAGIS creates a thread for each process of a network, and maintains a producer-consumer buffer for each channel. Production of data is non-blocking whereas consumption from an empty stream is blocking. As the reader can observe, this mechanism shares many similarities with the Unix process pipelining model.

Like most component-based frameworks, PAGIS processes are described in terms of *ports*. Ports define a communication contract with a process in the same way classes define interfaces for objects in object-oriented languages. In this way, applications are described by connecting ports through channels. PAGIS includes an API, called PNAPI (Process Network API), that provides several useful abstractions for describing applications in terms of process networks. Additionally, it offers a graphical tool for visually creating, composing and executing process networks.

PAGIS allows a process network to be supplied with Grid behavior by means of *metalevel programming*. Conceptually, metalevel programming divides an application into a *base* level, composed of classes and objects implementing its functional behavior, and a *meta* level, consisting of *metaobjects* that reify elements of the application at runtime –mostly method invocations– and perform computations on them. Figure 3.6 illustrates the basics of metalevel programming. Base level objects *ObjectA* and *ObjectB* have been both assigned two different metaobjects. As a consequence, *MetaA* receives all method invocations sent from *ObjectA* and redirects them to the target's metaobject (in this case *MetaB*), which actually carries out the invocations. The labels in bold represent the phases of a method invocation in which customized user actions can be associated.

PAGIS introduces the *MetaComputation* metaobject, specially designed to represent a running process network as one single structure. Users can then materialize complex Grid functionality by attaching metaobjects[2] to *MetaComputation* metaobjects. For example, one might implement

[2]Strictly speaking, these are meta-metaobjects, since they intercept method calls performed by other metaobjects.

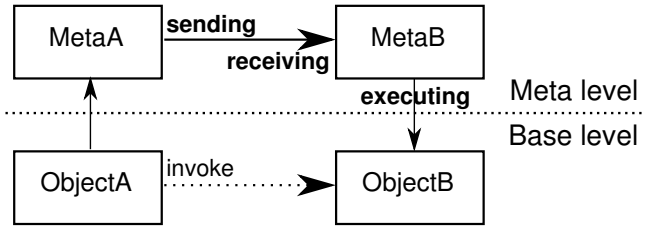


Figure 3.6: Overview of metalevel programming

a custom metaobject for transferring certain method invocations to a remote metaobject, thus achieving load balancing. Similarly, a metaobject that monitors and records the various runtime aspects of an application can be easily implemented by logging information such as timing, source and destination objects, amongst others, prior to method redirection.

The gridification scheme proposed by PAGIS is indeed interesting, since it allows to furnish ordinary applications (i.e. the base level) with Grid-dependent behavior (i.e. the meta level) without affecting its source code. The only requirement is that those applications are appropriately transformed so that they are structured as a process network. Similar to GridAspecting, a PAGIS application (i.e. a process) is specified at a task level of granularity.

3.1.8 ProActive

ProActive (Baduel et al., 2006) is a Java-based middleware for object-oriented parallel, mobile and distributed computing. It includes an API that isolates many complex details of the underlying communication and reflection Java APIs, on top of which a component-oriented view is provided. This API also includes functionality to transform conventional Java classes to a ProActive application. The programming model featured by ProActive has also been implemented in C++ and Eiffel.

A typical ProActive application is composed of a number of mobile entities called *active objects*. Each active object has its own thread of control and an entry point, called the *root*, by which the object functionality can be accessed from ordinary objects. Active objects serve methods calls issued from other active/ordinary objects, and also request services implemented by other local or remote active objects. Method calls sent to active objects are synchronized based on the *wait-by-necessity* mechanism, which transparently blocks the requester until the results of a call are received. At the ground level, this mechanism relies on meta programming techniques similar to that of PAGIS, thus it is very transparent to the programmer.

A Java Virtual Machine (JVM) participating in a computation can host one or more *nodes*. A node is a logical entity that groups and abstracts the physical location of a set of active objects. Nodes are identified through a symbolic name, typically a Uniform Resource Locator (URL). Therefore, active objects can be programmatically attached/detached from nodes without the need for manipulating low-level information like networks addresses or ports. Similarly, active

objects can be sent for execution to remote JVMs by simply assigning them to a different "container" node.

Standard Java classes can be easily transformed into active objects. For example, let us assume we have a class named *C*, which exposes two methods *foo* and *bar*, with return type *void* and *double*, respectively. The API call:

```
ProActive.newActive("C", args, "rmi://www.exa.unicen.edu.ar/node");
```

creates –by means of RMI– a new active object of type C on the node *node*. Further calls to either *foo* or *bar* are asynchronously handled by ProActive, and any attempt to read the result of an invocation to *bar* blocks the caller until the result is computed. In a similar way, the API can be used to straightforwardly publish an active object as a SOAP-enabled Web Service.

Another interesting feature provided by ProActive is the notion of *virtual nodes*. The idea behind this concept is to abstract away the mapping of active objects to physical nodes by eliminating from the application code elements such as host names and communication protocols. Each virtual node declared by the application is identified through a plain string, and mapped to one or a set of physical nodes by means of an external XML deployment descriptor file. As a consequence, the resulting application code is independent of the underlying execution platform and can be deployed on different Grid settings by just modifying its associated deployment descriptor file.

There are, however, some code conventions that programmers must follow before gridifying an ordinary Java class as an active object. First, classes must be serializable, and include a default constructor (i.e. with no arguments). Second, the result of a call to a non-void method should be placed on a local variable for the wait-by-necessity mechanism to work. Return types for non-void methods should be replaced by system-provided wrappers accordingly. In our example, we have to replace the return type in the *bar* method with a ProActive API class that wraps the *double* Java primitive type.

3.1.9 Satin

Satin (van Nieuwpoort et al., 2005a) is a Java framework that lets programmers to easily parallelize applications based on the divide and conquer paradigm. The ultimate goal of Satin is to free programmers from the burden of modifying and hand-tuning applications to exploit a Grid setting. Satin is implemented on top of Ibis (van Nieuwpoort et al., 2005b), a programming environment whose goal is to provide an efficient Java-based platform for Grid programming. Ibis consists of a highly-efficient communication library, and a variety of programming models, mostly for developing applications as a number of components exchanging messages through messaging protocols like Java RMI and MPI.

Satin extends Java with two primitives to parallelize single-threaded conventional Java programs: *spawn*, to create subcomputations (i.e. divide), and *sync*, to block execution until the results from subcomputations are available. Methods considered for parallel execution are identified by means

of *marker interfaces* that extend the *satin.Spawnable* interface. Furthermore, a class containing spawnable methods must extend the *satin.SatinObject* class and implement the corresponding marker interface. In addition, the result of the invocation of a spawnable method must be stored on a local variable. The next code shows the Satin version of a simple recursive solution to compute the k^{th} Fibonacci number:

```
interface IFibMarker extends satin.Spawnable{
  public long fibonacci(long k);
}

class Fibonacci extends satin.SatinObject implements IFibMarker{
 public long fibonacci(long k){
    if (k < 2)
      return k;
    // The next two calls are automatically spawned,
    // because "fibonacci" is marked in IFibMarker
    long f1 = fibonacci(k - 1);
    long f2 = fibonacci(k - 2);
    // Execution blocks until f1 and f2 are instantiated
    super.sync();
    return f1 + f2;
  }
  static void main(String[] args){
    ...
    Fibonacci fib = new Fibonacci();
    // also spawned
    long result = fib.fibonacci(k);
    // Blocks the main application thread
    // until a result is obtained
    fib.sync();
    ...
  }
}
```

After indicating spawnable methods and inserting appropriate synchronization calls into the application source code, the programmer must feed a compiled version of the application to a tool that translates, through Java bytecode instrumentation, each invocation to a spawnable method into a Satin runtime task. For example, in the code shown above, a task is generated for every single call to the *fibonacci* method.

Since each task represents the invocation (recursive or not) to a spawnable method, their granularity is clearly smaller than the granularity of tasks like the ones supported by GridAspecting or PAGIS. A running application may therefore have associated a large number of fine-grained

tasks, which can be executed on any machine. For overhead reasons, most tasks are processed on the machine in which they were created. In order to efficiently run gridified programs, Satin uses a task execution scheme based on a novel load-balancing algorithm called Cluster-aware Random Stealing (CRS). With CRS, when a machine becomes idle, it attempts to steal a task waiting to be processed from a remote machine. Finally, intra-cluster steals have a greater priority than wide-area steals. This policy fundamentally aims at saving bandwidth and minimizing the latencies inherent to slow wide-area networks.

3.1.10 XCAT

XCAT (Gannon et al., 2005) is a component-based framework for Grid application programming built on top of Web Services technologies. XCAT applications are created by connecting distributed components (OGSA Web Services) that communicate either by SOAP messaging or an implicit notification mechanism. XCAT is compliant with Common Component Architecture (CCA) (Armstrong et al., 1999), a specification whose goal is to come out with a reduced set of standard interfaces that a high-performance component framework should provide/-expect to/from components in order to achieve easy distributed component composition and interoperability.

XCAT components are connected by *ports*. A port is an abstraction representing the interface of a component. Ports are described in Scientific Interface Definition Language (SIDL), a language for describing component operations in terms of the data types often found in scientific programs. SIDL defines two kinds of ports: *provides* ports, representing the services offered by a component, and *uses* ports, which describes the functionality a component might need but is implemented by another component. Furthermore, XCAT provides ports can be implemented as OGSA Web Services. XCAT uses ports can connect to any typed XCAT provides port (i.e. those described in SIDL) but also to any OGSA-compliant Web Service.

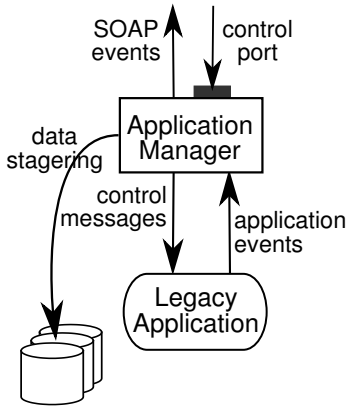


Figure 3.7: XCAT application managers

XCAT allows scientific legacy applications to be deployed as components without code modification using the concept of *application manager* (see Figure 3.7). Basically, each legacy application is wrapped with a generic component (application manager) responsible for managing and monitoring the execution of the application, and staging the necessary input and output data. The manager also serves as a forwarder for events taking place inside the wrapped application such as file creation, errors, and execution finalization/crash. Applications managers can be connected to each other, and have one special port by which standard components can control them. It is worth noting that legacy applications have, in general, a large granularity. As a consequence, XCAT shares some of the limitations of GEMLCA and GRASG with respect to granularity of gridified applications.

3.2 A Taxonomy of Gridification Approaches

Table 3.1 summarizes the main characteristics of the approaches described in the previous sections. To better understand the structure of the table, the reader should recall the analogy between the Grid and the electrical power grid discussed in Chapter 2.

Basically, each row of the table represents a "wall socket" by which applications are gridified and connected to the Grid. The "Appliance type" column symbolizes the kind of applications supported by the gridification process, whereas the "Grid-aware appliance" column briefly describes the new anatomy of applications after passing through the gridification process. In addition, lower-level Grid technologies upon which each approach is built are also listed. Finally, the discussion centers on the different approaches to gridification (i.e. the "plugging techniques") according to the taxonomies presented in the next subsections.

In particular, taxonomies of subsections 3.2.1 and 3.2.2 describe, from the point of view of source code modification, the different ways in which an ordinary application can be affected by the gridification process. The taxonomy presented in subsection 3.2.3 represents the observed granularity levels to which applications are Grid-enabled. Finally, the taxonomy included in subsection 3.2.4 categorizes the approaches according to the kind of Grid resources they aim to virtualize. These taxonomies are simple, but comprehensive enough to cover the various aspects of gridification.

3.2.1 Application Reengineering

The application reengineering taxonomy defines the extent to which an application must be manually modified in order to obtain its gridified counterpart. In general, the static anatomy of every application can be described as a number of *compilation units* combined with a certain *structure*. Compilation units are programming language-dependent pieces of software (e.g.

Wall socket	Appliance type	Plugging technique highlights	Grid-aware appliance	Underlying technologies
GEMLCA	Binary executable	The user must specify the interface of his/her application (XML file)	Globus-wrappered binary executable	Web Services; Globus
GrADS	C application (MPI-based if explicit migration is to be used)	Instruction insertion if using SRS	Globus-wrappered binary executable	Web Services; Globus; NWS; MPI
GRASG	Binary executable	Hand-tuning of applications through Perl/shell scripting	Binary executable interfaced through a *JES* Web Service	SOAP-based Web Services; Globus
Grid-Aspecting	Task-parallel Java application	Manual task decomposition and Grid concerns (aspects) implementation	Multi-threaded, aspect-enhanced Java application	AspectJ
GriddLeS	Stream-based binary executable	Transparent overloading of system libraries implementing file/sockets operations across the Grid	Globus-wrappered binary executable (*component*)	Globus; GridFTP
Ninf-G	C/Fortran application	Users decompose applications into client/server parts, and connect them using GridRPC calls	Client and server-side binary executables	Globus; GridRPC
PAGIS	Java application	Users identify components and assemble them through *channels* to build *process networks*	Binary executable (*process*)	-
ProActive	Java application	Source code conventions; proxy-based wrapping	Active object	SOAP-based Web Services; RMI
Satin	Divide and conquer Java application	Source code conventions; bytecode instrumentation enabling recursive method calls to be *spawnable*	Single-threaded, parallel Java application	Ibis
XCAT	Binary executable	The user must specify the interface of his/her executable in SIDL	Binary executable interfaced through an *application manager*	Web Services

Table 3.1: Summary of gridification tools

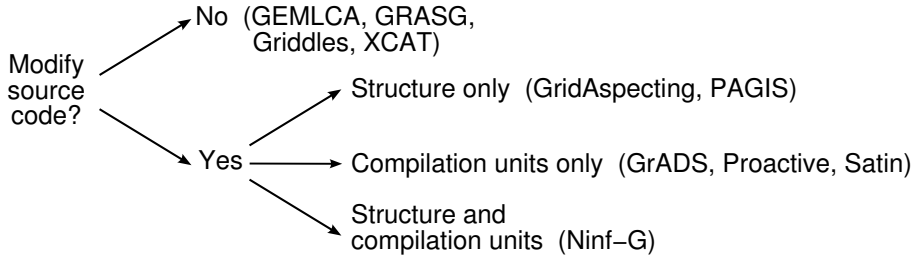


Figure 3.8: Application reengineering taxonomy

Java classes, C and Perl modules, etc.) assembled together to form an application. Usually, a compilation unit corresponds to a single source code file. Furthermore, the way compilation units are combined determines the structure of the application (e.g. the class hierarchy of a Java application, the dependence graph between functions within a C program). According to Figure 3.8, the anatomy of a conventional application might be altered in the following manners:

- *Structure only*: Some approaches alter the internal structure of the application, restructuring it in such a way some of its constituent parts are reorganized. For example, GridAspecting requires the user to identify tasks within the application that potentially can be executed concurrently. Similarly, the PAGIS framework requires to restructure applications as a set of components exposing and invoking services through well-defined interfaces. However, in both cases, the user code originating these tasks and components practically remains unchanged. In general, this procedure makes the appearance of the original application significantly different from that of the Grid-aware application, but the pure implementation code is practically the same. In other words, even though the code within compilation units may slightly change, the focus of structure modification is on redesigning the application instead of rewriting it (i.e. internally modify methods/procedures).

  Structure modification is a very common approach to gridification among template-based Grid programming frameworks. With these frameworks, the user adapts the structure of his/her application to a specific template implementing a recurring execution pattern defined by the framework. For example, JaSkel (Ferreira et al., 2006) is a Java framework for developing parallel applications that provides a set of abstract classes and a skeleton catalog, which implements interaction paradigms such as farm, pipeline, divide and conquer and heartbeat templates. Another example is MW (Goux et al., 2000), a framework based on the popular master-worker paradigm for parallel programming.

- *Compilation units only*: Conversely, other approaches alter only some compilation units of the application. For instance, ProActive and Satin require the developer to modify certain methods within the application to make them compliant to a specific coding convention. But, in both cases, the class hierarchy of the application is barely modified. A taxonomy of gridification techniques for single compilation units is presented in the next subsection.

  Examples of compilation unit modification are commonly found in the context of distributed programming. For instance, this technique is frequently employed when a single-

machine Java application is adapted to use a distributed object technology such as RMI or CORBA. Some of the (formerly local) objects are explicitly distributed on different machines and looked up by adding specific API calls inside the application code. However, the behavioral relationships between those distributed objects do not change. Additionally, similar examples can be found in distributed procedural programming using technologies such as MPI or RPC.

- *Structure and compilation units*: Of course, gridification methods may also modify both the structure and compilation units of the application. For example, Ninf-G demands the developer to split an application into client and server-side parts and then to modify the client so as to remotely interact with the server(s). Notice that the internal structure of the application changes dramatically, since a single server part may contain code combining the functions originally placed at different compilation units. Overall, not only the ordinary application is refactored by creating many separate programs, but also several modifications to some of the original functions are introduced.

Intuitively, the first technique enables the user to perform modifications at a higher level of abstraction than the second one. Users are not required to provide code for using Grid functionality and deal with Grid details, but have to change the application shape. As a consequence, the application logic is not significantly affected after gridification. In principle, the most undesirable technique is by far to modify the structure and the compilation units of an application, since not only the application shape is changed, but also the nature of its code. However, it is very difficult to determine whether a technique is better than the others, as the amount of effort necessary to gridify an application with either of the three approaches depends on its complexity/size, the amount/type of modifications imposed by the gridification method for restructuring and/-or rewriting the application, the programming language, and the particular Grid setting and underlying technologies being used for application execution.

3.2.2 Compilation Unit Modification

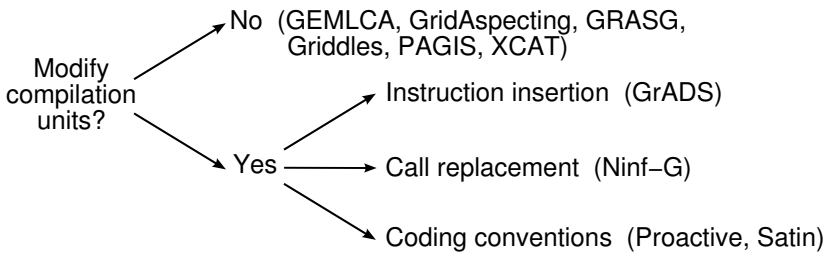


Figure 3.9: Compilation unit modification taxonomy

The compilation unit modification taxonomy determines how applications are altered after gridification with respect to modification of theirs compilation units. As shown in Figure 3.9, we

can broadly identify the following categories:

- *Instruction insertion*: The most intuitive way to gridify is, as it name indicates, by manually inserting instructions implementing specific Grid functionality at proper places within the code. A case of instruction insertion arises in GrADS when the user wants to explicitly control application migration and data staging. A clear advantage of this technique is that the programmer can optimize his/her application at different levels of granularity to produce a very efficient Grid application. However, in most cases, the application logic is literally mixed up with Grid-related code, thus making maintainability, legibility, testing and portability to different Grid platforms hard.

  There are many Grid middlewares that require users to employ instruction insertion when gridifying compilation units. For example, JavaSymphony (Jugravu and Fahringer, 2005) is a programming model whose purpose is to simplify the development of performance-oriented, object-based Grid applications. It provides a semi-automatic execution model that deals with migration, parallelism and load balancing of applications, and at the same time allows the programmer to control –via instruction insertion– such features as needed. Other examples are Javelin 3.0 (Neary and Cappello, 2005) and GridWay (Huedo et al., 2004), two platforms for deployment and execution of CPU-intensive applications which require users to modify applications in order to exploit job checkpointing and parallelization.

- *Call replacement*: A very common technique to gridify compilation units is by replacing certain groups of sequential instructions by appropriate calls to the underlying middleware API. Such instructions may range from operations for carrying out interprocess communication to code for manipulating data. Unlike the previous case, call replacement puts more emphasis on replacing certain pieces of conventional code by Grid-aware code instead of inserting new instructions throughout the user application.

  Call replacement assumes users know what portions of their code should be replaced in order to adapt it for running on a particular Grid middleware. Nevertheless, in order to help users in doing this task, Grid middlewares usually offer guidelines tutoring users on how to port their applications. For instance, gridifying with the Globus platform involves replacing socket-based communication code by calls to the Globus Input/Output service, transforming conventional data copy/transfers operations by GridFTP operations, and finally replacing all resource discovery instructions (e.g. for obtaining available execution nodes) by calls to the MDS-2 service.

  Clearly, call replacement is a form of gridification suitable for users who are familiar with the target middleware API. Users not having a good understanding of the particular API to be used may encounter difficult to port their applications to the Grid. Another drawback of the approach is that the resulting code is highly-coupled with a specific Grid API, thus having many of the problems suffered by instruction insertion. These issues are partially solved by toolkits that attempt to offer a comprehensive, higher-level programming API on

top of middleware-level APIs (e.g. Java CoG Kit, GAT). However, developers are forced to learn yet another programming API. Indeed, Grid toolkits help alleviating developer pain caused by call replacement, but certainly they are not the cure.

- *Coding conventions*: This technique is based on the idea that all the compilation units of an ordinary application must obey certain conventions about their structure and coding style prior to gridification. These conventions allow tools to properly transform a gridified application into one or more middleware-level execution units. For example, ProActive requires application classes to extend the Java *Serializable* interface. Moreover, Satin requires that the result of any invocation to a recursive method is placed on a variable, rather than accessing it directly (e.g. pass it on as an argument to another method). Unlike instruction insertion and call replacement, the gridified code is in general not tied to any specific Grid API or library.

  To provide an illustrative example in the context of conventional software, we could cite JavaBeans (Englander, 1997), a widely-known specification from Sun that defines conventions for writing reusable software components in Java. In order to operate as a JavaBean, a class must follow conventions about method naming and behavior. This, in turn, enables easy graphical reuse and composition of JavaBeans to create complex applications with little implementation effort.

It is worth pointing out that using any of the above techniques does not automatically exclude from using the others. In fact, they usually complement each other. For example, Satin, though it is focused on gridifying by imposing coding conventions, requires programmers to coordinate several calls to a spawnable computation within a method by explicitly inserting special synchronizing instructions. Furthermore, it is unlikely that an application that has been adapted to use a specific Grid API (e.g. GridFTP in the case of Ninf-G) does not include user-provided instructions for performing some API initialization or disposal tasks.

3.2.3 Gridification Granularity

Granularity is a software metric that attempts to quantify the size of the individual components[3] that make up a software system. Large components (i.e. those including much functionality) are commonly called *coarse-grained*, while those components providing little functionality are usually called *fine-grained*. For example, with service-oriented architectures (Huhns and Singh, 2005), applications are built in terms of components called *services*. In this context, component granularity is determined by the amount of functionality exposed by services, which may range from small (e.g. querying a database) to big (e.g. a facade service to a travel business).

[3]The term "component" refers to any single piece of software included in a larger system, and should not be confused with the basic building blocks of the component-based programming model

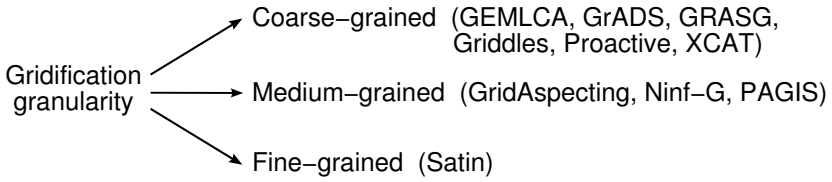


Figure 3.10: Gridification granularity taxonomy

Notwithstanding, granularity is usually associated with the size of application components from a user's point of view, the concept can also be applied to get an idea of how granular the runtime components of a gridified application are. In this sense, *gridification granularity* is defined as the granularity of the individual components that constitute an executing gridified application from the point of view of the Grid middleware. Basically, these Grid-enabled components are execution units like jobs or tasks to which the Grid directly provides scheduling and execution services. Note that "conventional" granularity does not necessarily determines gridification granularity. Clearly, this is because the former is concerned with the size of components *before* an application is transformed to run on a Grid setting. For example, during gridification, a single coarse-grained service might be partitioned into several more granular services to achieve scalability.

Like conventional granularity, gridification granularity takes continuous values ranging from the smallest to the largest possible component size. As shown in the taxonomy of Figure 3.10, the spectrum of gridification granularities is divided into three discrete values:

- *Coarse-grained*: A running application is composed of a number of "heavy" execution units. Typically, the application execution is handled by just one runtime component. This level of granularity usually results from employing solutions such as GEMLCA, GRASG, GriddLes and XCAT, which adapt the executable of an ordinary application to be executed as a single Grid-aware job. At runtime, a job behaves like a "black box" that receives a pre-determined set of input parameters (e.g. numerical values, files, etc.), performs some computation, and returns results back to the executor. A similar case occurs with compiled versions of applications gridified with GrADS.

  Coarse-grained gridification granularity suffers from two major problems. On one hand, the application is treated by the middleware as a single execution unit. Therefore, unless refactored, it may not be possible for the individual resource-consuming parts of a running application to take advantage of mechanisms such as distribution, parallelization or scheduling to achieve higher efficiency. On the other hand, the middleware sees a running application as an indivisible unit of work. As a consequence, some of its portions that might be dynamically reused by other Grid applications (e.g. a data mining algorithm) cannot be discovered or invoked.

  To a lesser extent, ProActive applications can also be considered as coarse-grained. An ordinary Java application (i.e. its main class and helper classes) is gridified by transforming

it to a self-contained active object. The user sees the non-gridified application as composed of a number of (medium-grained) objects. On the other hand, ProActive sees the gridified application as one big (active) object. When executing the application, the ProActive runtime performs scheduling and distribution activities on active objects rather than on plain objects. Nevertheless, ProActive is more flexible than the other approaches in this category, since it lets developers to explicitly manage mobility inside an active object, invoke methods from other active objects and externalize the methods implemented by an active object.

- *Medium-grained*: The running application has a number of execution units of moderate granularity. Systems following this approach are GridAspecting, Ninf-G and PAGIS. In the former and latter case, the user identifies those tasks within the application code that can be executed concurrently. Then, they are mapped by the middleware to semi-granular runtime task objects. Similarly, a running Ninf-G application is composed of several IDL-interfaced processes that are distributed across a network. Unlike GridAspecting and PAGIS, this approach affords an opportunity for dynamic component invocation, as a Ninf-G application might perform calls to the functions exposed by the components of another Ninf-G application.

- *Fine-grained*: This category represents the gridification granularity associated to runtime components generated upon the invocation of a method/procedure. A representative case of fine-grained granularity is Satin. Basically, a middleware-level task is created after every single call to a spawnable method, regardless of whether calls are recursive or not. From the application point of view, there is a better control of parallelism and asynchronism. However, a running application may generate a large number of tasks that should be efficiently handled by the underlying middleware. This fact suggests the need for a runtime support providing sophisticated execution services smart enough to efficiently deal with task scheduling and synchronization issues.

It is worth noting that, in some cases, the user may indirectly adjust (e.g. by refactoring code) the gridification granularity to fit specific application needs. For example, a set of medium--grained tasks could be grouped into one bigger task in order to reduce communication and synchronization overhead. Conversely, the functionality performed by a task could be decomposed into one or more tasks to achieve better parallelism. Nonetheless, this process can be cumbersome and sometimes counterproductive. For example, ProActive applications can be restructured by turning standard objects into active objects, but then the programmer must explicitly provide code for handling active object lookup and coordination. Similarly, gridification granularity of Ninf-G applications can be reduced by increasing the number of server-side programs. However, this could cause the application to spend more time communicating than doing useful computations.

3.2.4 Resource Harvesting

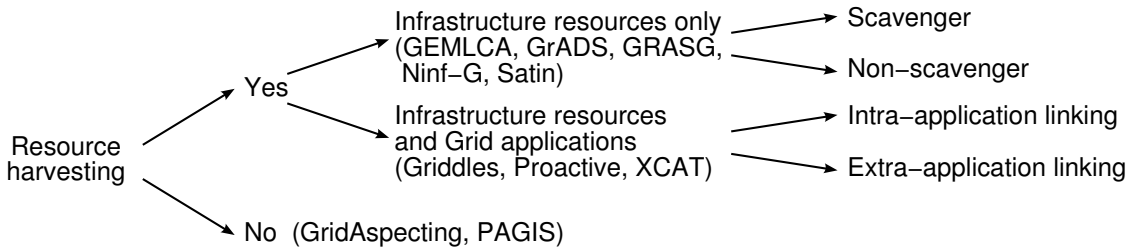


Figure 3.11: Resource harvesting taxonomy

The resource harvesting taxonomy describes, in a general way, the kind of Grid resources to which access is made transparent by each gridification method. The utmost goal of Grid Computing, as explained in the previous chapter, is to virtualize distributed resources so they can be transparently used and consumed by ordinary applications. Certainly, gridification tools play a fundamental role in achieving such a transparency. The resource harvesting taxonomy is depicted in Figure 3.11.

Surprisingly, some gridification methods do not pursue resource virtualization. Specifically, solutions such as GridAspecting or PAGIS aim at preserving the integrity of the application logic during gridification and make them independent of a specific Grid platform or middleware. In this way, users have the flexibility to choose the runtime support or middleware that better suits their needs. However, as these approaches do not offer facilities for using Grid services, the burden of providing the "glue" code for interacting with the Grid is entirely placed on the application developer, which clearly demands a lot of programming effort.

Most gridification methods, however, provide some form of Grid resource leveraging, along with a minimal or even no effort from the application developer. Basically, these are integrated solutions that offer services for gridifying ordinary applications as well as accessing Grid resources. Depending on the type of resource they attempt to virtualize, these methods can be further classified as:

- *Infrastructure resources only*: Applications resulting from applying the gridification process are not concerned with providing services to other Grid applications. Applications are simply ported to the Grid to transparently leverage middleware-level services (e.g. resource brokering, load-balancing, mobility, scheduling, parallelization, storage management, etc.) that virtualize and enhance the capabilities of computational resources like processing power, storage, bandwidth, and so forth. Moreover, some approaches are more focused on harnessing idle CPU power (the so called "scavengers"; GrADS, Ninf-G, Satin), whereas others also include simple abstractions and easy-to-use services to deal with data management on the Grid (GEMLCA, GRASG). In any case, the emphasis is solely put on taking

advantage of Grid resources, rather than using Grid services *and* services implemented by other Grid applications.

- *Infrastructure resources and Grid applications*: The goal of these approaches is to simplify the consumption of both Grid services and functionality offered by gridified applications. At the middleware level, gridified applications are treated just like any other individual Grid resource: an entity providing special capabilities that can be used/consumed by other applications by means of specialized Grid services. Note that this is a desirable property for a gridification tool, since reusing existing Grid applications may improve application modularity and drastically reduce development effort (Atkinson et al., 2005).

  Linking together Grid applications requires the underlying middleware to provide, in principle, mechanisms for communicating applications. These mechanisms may range from low-level communication services such as those implemented by GriddLes, to high-level, interoperable messaging services like SOAP. In addition, mechanisms are commonly provided to describe the interface of a gridified application in terms of the internal services that are made accessible to the outside, and also to discover existing Grid applications. For example, popular technologies for describing and discovering Grid applications are WSDL (W3C Consortium, 2007b) and UDDI (OASIS Consortium, 2004), respectively. Table 3.2 briefly compares the tools that support application linking by showing how they deal with application interface description, communication and discovery.

  There are basically two forms to connect applications: *extra-application* and *intra-application*. In the extra-application approach, existing Grid applications can be reused by combining and composing them into a new application. For example, XCAT conceives gridified applications as being indivisible components that can be combined –with little coding effort– into a bigger application, but no binding actions are ever carried out from inside any of these components. Another example of a gridification tool following this approach is GMarte (Alonso et al., 2006), a high-level Java API that offers an object-oriented view on top of Globus services. With GMarte, users can compose and coordinate the execution of existing binary codes by means of a (usually small) new Java application. On the other hand, in the intra-application linking approach, users are not required to implement a new "container" application, since binding to existing Grid applications is performed within the scope of a client application. GriddLeS and ProActive are examples of approaches based on intra-application linking.

Gridification approaches oriented towards consuming Grid resources are engaged in finding ways to make the task of porting applications to use Grid services easier. On the other hand, approaches seeking to effortlessly take advantage of Grid resources and existing applications generalize this idea by providing a unified view over Grid resources in which applications not only consume but also offer Grid services. It is important to note that this approach shares many similarities with the service-oriented model, where applications may act both as clients and providers of services. As discussed in Chapter 2, many global Grid standards such as OGSA

and WSRF have already embodied the convergence of service-oriented architectures and Grid Computing technologies.

Tool	Interface description	Communication protocol	Application discovery
GriddLeS	Implicit (file-based)	Sockets	No
ProActive	Explicit (WSDL)	SOAP	Yes (lookup by active object identifier)
XCAT	Explicit (WSDL)	SOAP or XML-based implicit notification	No

Table 3.2: Comparison between gridification tools leveraging both Grid resources and applications

3.3 Discussion

Table 3.3 summarizes the approaches discussed so far. Each cell of the table corresponds to the taxonomic value associated to a particular tool (row) with respect to each one of the taxonomies presented in the previous section.

There are approaches that let users to gridify applications without modifying a single line of code. Solutions belonging to this category take the application in their binary form, along with some user-provided configuration (e.g. input and output parameters and resource requirements), and wrap the executable code with a software entity that isolates the complex details of the underlying Grid. It is important to note that this approach has both advantages and disadvantages. On one hand, the user does not need to have a good expertise on Grid technologies to gridify his/her applications. Besides, applications can be plugged into the Grid even when the source code is not available. On the other hand, the approach results in extremely coarse-grained gridified applications, thus users generally cannot control the execution of their applications in a fine--grained manner. This represents a clear tradeoff between ease of gridification versus flexibility to control the various runtime aspects of a gridified application.

Tool	Application reengineering	Compilation unit modification	Gridification granularity	Resource harvesting
GEMLCA	No	No	Coarse-grained	Grid resources
GrADS	Yes (compilation units only)	Instruction insertion	Coarse-grained	Grid resources
GRASG	No	No	Coarse-grained	Grid resources
Grid-Aspecting	Yes (structure only)	No	Medium-grained	No
GriddLeS	No	No	Coarse-grained	Grid resources and applications
Ninf-G	Yes (structure and compilation units)	Call replacement	Medium-grained	Grid resources
PAGIS	Yes (structure only)	No	Medium-grained	No
ProActive	Yes (compilation units only)	Code conventions	Coarse-grained	Grid resources and applications
Satin	Yes (compilation units only)	Code conventions	Fine-grained	Grid resources
XCAT	No	No	Coarse-grained	Grid resources and applications

Table 3.3: Summary of gridification approaches

A remarkable result of the survey is the diversity of programming models existing among the analyzed tools: procedural and message passing (GrADS), AOP (GridAspecting, PAGIS), workflow-oriented (GriddLes), RPC (Ninf-G), component-based (PAGIS, ProActive, XCAT), object-oriented (ProActive, Satin), just to name a few. This evidences the absence of a widely-adopted programming model for the Grid, in contrast to other distributed environments (e.g. the Web) where well-established models for implementing applications are found (Johnson, 2005).

Another interesting result is the way technologies like Java, Web Services and Globus have influenced the development of gridification tools within the Grid. Specifically, many of the surveyed tools are based on Java or rely on Web Services, and almost all of them either build on top of Globus or provide some integration with it. Nevertheless, this result should not be surprising for several reasons. Java has been widely recognized as an excellent choice for implementing distributed applications mainly because of its "write once, run anywhere" philosophy, which promotes platform independence. Web Services technologies enable high interoperability across the Grid by providing a layer that abstracts clients and Grid services from network-related details such as protocols and addresses. Lastly, Globus has become the *de facto* standard toolkit for

implementing Grid middlewares, since it provides a continuous evolving and robust API for common low-level Grid functionality such as resource discovery and monitoring, job execution and data management.

3.4 Conclusions

Grid Computing promises users to effortlessly take advantage of the vast amounts and types of computational resources available on the Grid by simply plugging applications to it. However, given the extremely heterogeneous, complex nature inherent to the Grid, adapting an application to run on a Grid setting has been widely recognized as a very difficult task. In this sense, a number of technologies to facilitate the gridification of conventional applications have been proposed in the literature. In this chapter, some of the most related technologies to the purpose of this thesis have been reviewed.

Unfortunately, the approaches to gridification previously discussed only cope with a subset of the problems that are essential to truly achieving gridification, while not addressing the others. Namely, these problems are:

- **The need for modify and/or restructure applications.** Ideally, ordinary applications should be made Grid-aware without the need for manual code modification, adaptation or refactoring to use a Grid. Besides drastically reducing development effort, this would enable even the most novice Grid users to quickly and easily put their applications at work on the Grid.

  Even when the aim of the existing approaches is to ease gridification, they fail in making the task of porting applications to the Grid a transparent process. While the analyzed technologies do provide a two-step gridification process (see Section 2.3) in which programmers first concentrate on coding the logic of their applications and then on Grid-enabling them, this latter step still demands significant effort from developers in terms of application redesign and programming. This problem is evident in approaches such as GrADS, GridAspecting, Ninf-G, PAGIS and ProActive, and, to a lesser extent, Satin. This, in turn, make it difficult to fulfill desirable software quality attributes such as maintainability, testability, platform-independence and portability, among others.

- **Poor solutions to the ”ease of gridification versus tuning flexibility” tradeoff.** A number of toolkits avoid the problem of source code modification by simply adapting the binary version of an ordinary application to a Grid-aware one that, at runtime, is composed of just one single execution unit. In some cases (e.g. GEMLCA, GRASG) gridified applications are wrapped with Web Services technologies so as to achieve interoperability. However, this oversimplification of the gridification problem leads to another problem: it is not possible for the programmer to take advantage of mechanisms such as parallelization or distribution. For example, it is a good idea to execute individual CPU-bound parts of

an application concurrently at several host to improve performance. In addition, the mechanisms offered by these toolkits for application tuning are usually very limited and based on static parameter configuration or low-level scripting. On the other hand, approaches like the one followed by ProActive focus on providing a rich support for programmatically managing distribution, location and parallelization of application components but, again, the application being gridified must be explicitly modified. In conclusion, the gridification process should also take into account the runtime characteristics of the applications being gridified and provide mechanisms by which users easily adjust the granularity of application components, so as to produce Grid-aware applications that can be efficiently run across the Grid.

- **Gridified applications live "in their own bubble".** In many approaches (GEMLCA, GrADS, GRASG, Ninf-G, Satin), application after gridification are only concerned with using Grid resources while not interacting with other Grid applications at all. In other words, the gridification process is exclusively focused on enabling applications to take advantage of infrastructure resources, rather than producing Grid applications that provide services to and use the functionality of other Grid applications. However, in order to promote reusability of applications and collaboration across the Grid, conventional applications should be able to benefit from Grid resources as well as existing Grid applications, without any or eventually little coding effort, along with the higher interoperability levels required on the Grid.

  Recently, SOAs have emerged as an elegant approach to tackle down some of these problems. Under the SOA vision, the Grid is seen as a networked infrastructure of interoperating services representing both Grid resources and applications (Mauthe and Heckmann, 2005). SOAs provide the basis for *loose coupling*: interacting services know little about each other in the sense a single service discovers the necessary information to use external services (protocols, interfaces, location, and so on) in a dynamic fashion. This helps in freeing Grid developers from explicitly providing code for connecting applications together and accessing resources from within an application. Moreover, SOAs enable application interoperability as they usually rely on standard and ubiquitous technologies. As a matter of fact, it is not clear where to draw the line between Grid Services and Web Service technologies (Stockinger, 2007). Furthermore, current Grid standards are actively promoting the use of SOAs and Web Services for materializing the next generation architectures and middlewares for the Grid (Atkinson et al., 2005).

The next chapter describes an approach to solve these problems.

Chapter 4

The GRATIS Approach

In the past few years, Grid Computing has witnessed a number of significant advances in solutions aimed at simplifying the process of porting software to the Grid. However, as discussed in the previous chapter, current approaches are some way off from being effective gridification methods, and present key problems that require attention (Mateos et al., 2007a).

First, integrating existing source code onto the Grid still requires to rewrite many portions of the application to include Grid-related code, thus negatively affecting maintainability, legibility and portability to different Grid platforms. This problem is addressed by gridification methods that take applications in their binary form, along with some user-provided configuration, and wrap the executables with a software entity that isolates the complex details of the Grid. However, this results in extremely coarse-grained Grid applications, thus users generally cannot control the execution of their applications in a fine-grained manner to make better use of Grid resources. Finally, existing approaches do not have fully acknowledged the benefits that service-oriented technologies bring for Grid Computing, mainly in terms of application reusability across the Grid.

This thesis proposes a new gridification method that solves the problems mentioned above called GRidifying Applications by Transparent Injection of Services (GRATIS). Central to GRATIS is the concept of Dependency Injection (DI) (Johnson, 2005), a form of the Inversion of Control notion found in object-oriented frameworks. In DI, objects providing certain services are transparently injected into objects that require these services. GRATIS exploits this concept by allowing developers to inject Grid services into their ordinary applications with little effort. In essence, GRATIS aims at minimizing the requirement of source code modification when porting conventional applications to the Grid, and at the same time providing easy-to-use mechanisms to effectively tune Grid applications. Following the two-step gridification methodology, the idea is to let developers to focus first on implementing, testing and optimizing the functional code of their applications, and then to Grid-enable them. GRATIS promotes separation between application logic and Grid behavior by non-intrusively injecting all Grid-related functionality needed by the application at the second step of the gridification process.

It is worth pointing out that GRATIS is essentially a technology-agnostic gridification method. However, in order to provide a down-to-earth description of the approach, the contents of this chapter will be oriented towards describing GRATIS in the context of JGRIM (Mateos et al., 2008), a Grid middleware for gridifying Java-based, component-oriented applications. Basically, JGRIM provides a software artifact that supports the gridification method that will be presented in the sections that follow. Chapter 6 will explain the design and implementation of this middleware.

The chapter is structured as follows. The next section presents an overview of the GRATIS method by giving an account of its purpose, scope and related concepts, and introduces much of the terminology that will be used in the rest of the chapter. Then, Section 4.2 examines the concept of Dependency Injection. Later, Section 4.3 explains the gridification process of GRATIS and how this process is supported in the JGRIM middleware. Specifically, the section starts by introducing the required notions and describing the anatomy of both ordinary and gridified applications, and then shows a concrete gridification example. Finally, Section 4.4 presents the conclusions of the chapter.

4.1 Aims and Scope

GRATIS is a novel approach for easily creating and deploying applications on service-oriented Grids. The utmost goal of GRATIS is to make materialization and deployment of Grid applications simpler, by letting developers to focus on the development and testing of application logic without worrying about common Grid-related functionality such as resource discovery, service invocation and execution management. In other words, the goal is to permit applications to discover and efficiently use the vast amount of services offered by the Grid without the need to explicitly provide code for either finding or directly accessing these services from within the application logic. As the reader can see, GRATIS belongs to the category of two-step gridification methods.

It is important to note that GRATIS does not aim at providing yet another infrastructure for dealing with application and resource management on the Grid. Instead, its purpose is to provide a layer whereby ordinary applications are transformed to applications which are furnished with specialized *middleware-level components* that take advantage of existing Grid services, but minimizing as much as possible the users' knowledge required to carry out this transformation. From now on, we will refer to such transformed applications as *gridified* or *Grid-aware* applications. In addition, for the sake of simplicity, middleware-level components will be called m-components.

Figure 4.1 depicts an overview of GRATIS. As shown, GRATIS aims at transparently leveraging existing infrastructure Grid services by adding an intermediate middleware layer that enables

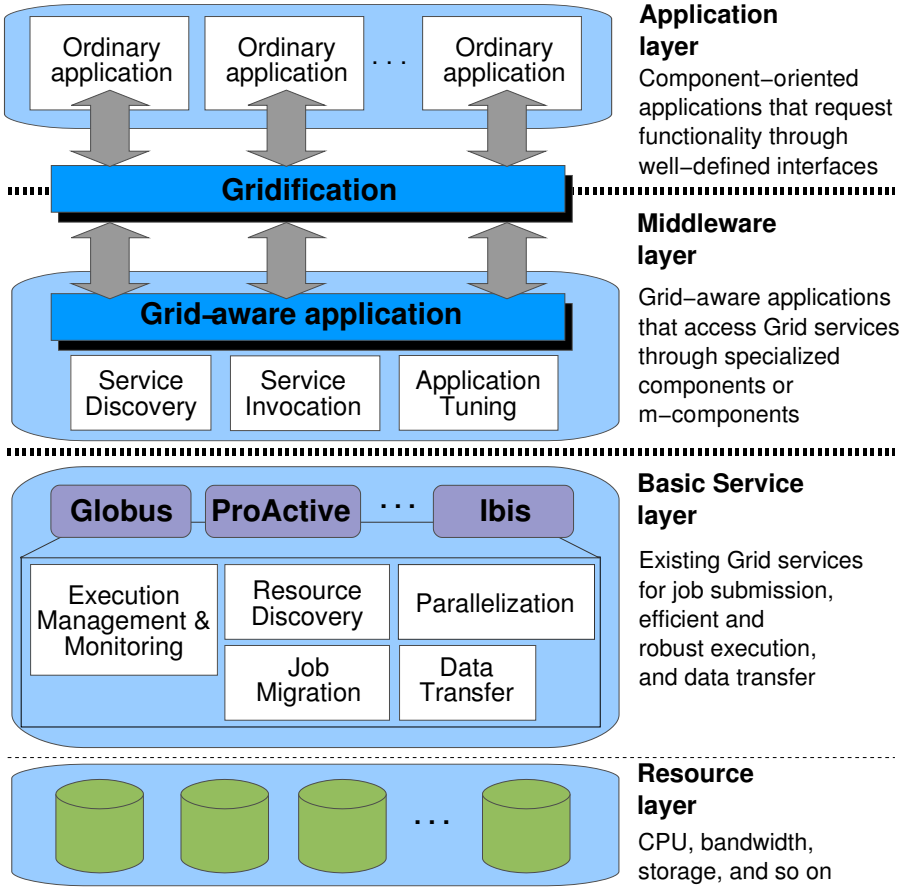


Figure 4.1: The GRATIS approach: a layered view

component-based applications to seamlessly use Grid services. Specifically, GRATIS considers four distinct layers at which Grid service provisioning is performed, namely:

- *Resource layer*: It simply represents the physical infrastructure of a Grid, given by resources such as computing nodes, networking capabilities and storage systems, along with the necessary low-level protocols to interact with them. Strictly speaking, this layer is basically a provider of resources –and not services– to the upper layers. Note that, as here defined, the resource layer encompasses both the Fabric and Resource and Connectivity layers of the Grid architecture presented in Section 2.1.

- *Basic Service layer*: This layer provides, by means of concrete resource management systems such as Globus, Condor or Ibis, a catalog of services whose main goal is to make a smart use of Grid resources. Services at this layer include load balancing, execution parallelization, resource brokering, fault tolerance, and so on. Basically, they represent sophisticated Grid functionality that is accessible to applications through specific protocols and APIs.

- *Middleware layer*: The Middleware layer is composed of a number of entities that roughly act as a glue between conventional applications and the Grid. The solely purpose of

these entities is to isolate applications from technology-related details for accessing the Basic Service layer (i.e. the aforementioned protocols and APIs) by means of specialized components or *metaservices*. A metaservice can be understood as a representative of a set of interrelated concrete services, that is, those providing similar Grid functionality. Examples of metaservices include:

  - *Service Discovery*: It represents the services within the Basic Service layer that perform service lookup on a Grid. Service discovery metaservices can talk, for example, to a UDDI registry or a MDS-2 Globus service in order to find a list of required Web Services, and then present this information to the Application layer in a Grid--independent format. In other words, the purpose of this m-component is to hide all concrete lookup services behind a generic, technology-neutral *lookup(serviceInterface)* primitive.
  - *Service Invocation*: Once one or more instances of a required service have been discovered, interaction with these instances comes next. In this way, it may be necessary to make use of different protocols, datatype formats, binding parameters, and so on. Typically, these elements are specified in the so-called service descriptors (e.g. a WSDL document), which represent any information that is necessary to contact an individual service instance. To sum up, the goal of the Service Invocation m-component is to isolate applications from the technologies involved in using Grid services or, in other words, to provide a generic *call(serviceDescriptor)* primitive.
  - *Application Tuning*: The GRATIS approach takes as input component-oriented applications, where components do not share any state and communicate through well-defined interfaces, and works by associating tuning services to certain requests from an application-level component to another. For example, an invocation performed on a time-consuming operation of a component may be submitted to a Globus or a Ibis environment, thus achieving better performance, scalability and reliability. In summary, the purpose of the Application Tuning metaservice is to make the execution of individual calls to component operations more efficient by leveraging (and potentially enhancing) underlying services for job execution, load balancing, parallelization and mobility.

- *Application layer*: According to Figure 4.1, ordinary applications are composed of a number of components, each one clearly described through an interface, that is, a list of one or more operations and their corresponding signature. After gridification, operation requests originated at the application level are handled by m-components, which are basically in charge of dealing with service discovery and interaction within the Grid.

In opposition with the approaches discussed in the previous chapter, in which users have to explicitly alter the application to use Grid services, the aim of GRATIS is to non-intrusively "inject" Grid services into the logic of ordinary applications. The assumption that drives this

injection is that it is possible to associate Grid services to the various *dependencies* of individual software components. By definition, a component C_1 has a dependency to another component C_2 when C_1 explicitly uses any of the operations of C_2. By associating metaservices to dependencies, interacting components can indirectly benefit from Grid services without changing their internal implementation.

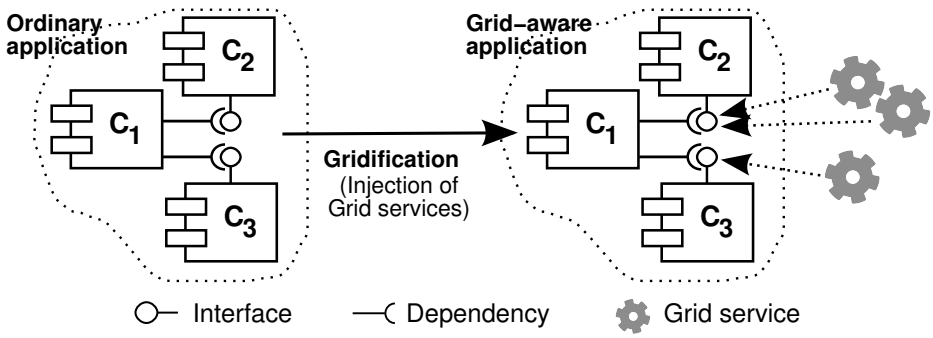


Figure 4.2: Component dependencies and Grid service injection

Figure 4.2 summarizes the concepts exposed so far. Components of an ordinary application interact with each other by means of their interfaces, thus establishing dependencies. Upon gridification, individual dependencies are associated with one or more metaservices, which customize the way a dependency behaves in a Grid setting. In the end, the gridified application will have injected into it one or more Grid services, which are concrete services providing non-functional Grid behavior to the application.

An interesting implication of dependencies is that they may also refer to external functionality for which an ordinary application does not provide an implementation. Specifically, an application component might declare a dependency to a missing component whose interface is known. As a matter of fact, a Grid often offers services not only for managing resources, but also for providing functional capabilities (e.g. algorithms, search engines, etc.) that play the role of third-party building blocks for creating new applications. Based on this fact, GRATIS distinguishes two types of Grid services:

- *Non-functional services*, which are those located at the Basic Service layer and represented by m-components. Non-functional services are characterized by a lack of a clear and standard interface to its capabilities, as they commonly represent abstract Grid concerns rather than explicitly-interfaced, callable services. Examples of this services are those providing parallelization, synchronization and coordination, mobility, load balancing and fault tolerance, among others.

- *Functional services*, which are basically a service interface to an already deployed Grid application that is external to the application being gridified. Similarly, gridified applications

may become themselves functional services if an interface describing their functionality is supplied. In this way, applications do not "live in their own bubble" after gridification, and a smooth correspondence with the SOA paradigm is achieved. Note that injecting functional services usually implies the injection of non-functional services as well (e.g. using security mechanisms when contacting functional services), but not the other way around.

The remainder of this chapter will be mostly concerned with describing how GRATIS injects m-components into conventional applications and, to a lesser extent, with the mechanisms for bridging these m-components with concrete Grid services, which will be explained in Chapter 6 in more detail. In this sense, the next section steps into making a deeper exploration of the relationships between component dependencies and Grid services.

4.2 Dependency Injection

Central to GRATIS is the concept of DI (Johnson, 2005). The idea behind this concept is to establish a level of abstraction between application components via public interfaces, and to remove dependency on components by delegating the responsibility for object creation and object linking to an application container or DI container. In other words, components only know each other's interfaces; it is up to the DI container to create and set (hence "inject") to a client component an instance of another component implementing a certain interface upon method calls on that interface.

By drawing a parallel with service-oriented software, the former component can be seen as the client requesting services, whereas the latter as one potential provider of these services. Here, the container would be the runtime platform in charge of binding clients to service providers. For example, a Web application might simply declare a dependency to a keyword-based search service. Then, the runtime platform in which the application is hosted (e.g. an application server) would be responsible for associating a concrete service for that dependency, such as a Web Service-based interface to the Amazon or the Google search engine.

4.2.1 Non-DI Applications: An Example

In the following paragraphs we will illustrate the concept of DI through an example. Let us suppose we have a Java application for listing books of a particular topic (`BookLister`) which fetches a remote text file where book information is stored. In addition, let us assume we are using the GridFTP protocol for transferring the file. The code implementing this application is very simple: setup a GridFTP connection to the remote site, transfer and parse the file, and then locally iterate the results in order to display book information. The code of our lister is:

```
import java.io.RandomAccessFile;
import java.util.Enumeration;
import java.util.List;
```

```
import org.globus.ftp.FileRandomIO;
import org.globus.ftp.GridFTPClient;

public class BookLister{
  public void displayBooks(String topic){
    // setup a GridFTP connection to [hostname:port]...
    GridFTPClient client = new GridFTPClient(hostname, port);
    RandomAccessFile raf = new RandomAccessFile("local-books.info");
    FileRandomIO sink = new FileRandomIO(raf);
    fileSize = client.getSize(remoteFile);
    client.extendedGet("remote-books.info", fileSize, sink, null);
    client.close();
    List books = parseBooks("local-books.info");
    Enumeration elems = books.elements();
    while (elems.hasMoreElements()){
      Book book = (Book)elems.nextElement();
      if (book.getTopic().equals(topic))
        System.out.println(book.getTitle() + ":" + book.getYear());
    }
  }
}
```

Now, if we want to use a completely different mechanism for retrieving book information such as querying a database or calling a Web Service, `displayBooks` method must be rewritten. Clearly, this is because the application code is designed to handle a specific type of information source, or a GridFTP server in this case. But there is more: depending on the way information is accessed, a different set of configuration parameters might be required (e.g. passwords, endpoints, URLs, etc.). In such a case, `BookLister` must also be modified to include the necessary instance variables and constructors/setters methods. This is rather undesirable, since a big part of the application would be tied to a specific file transfer technology.

4.2.2 A DI-based Solution

Next we present an alternative implementation of the above example that is coded according to the DI concept. The DI version of the listing component could include an interface (`BookSource`) by which `BookLister` accesses the book information, and several components implementing this interface for each form of fetching. Additionally, `BookLister` could expose a method `setSource(BookSource)` so that the container can inject the particular retrieval component being used. Note that BookLister now contains code only for iterating and displaying information, which is in fact pure application logic, but the code which knows how to obtain this information is placed on extra components. The new version of the example is:

```
public interface BookSource{
  public List getBooks(String topic);
}

public class GridFTPBookSource implements BookSource{
  ...
  public void sethostname(String hostname){
    this.hostname = hostname;
  }
  public void setport(int port){
    this.port = port;
  }
  public void setremoteFileName(int remoteFileName){
    this.remoteFileName = remoteFileName;
  }
  public List getBooks(String topic){
    /**
     * 1) setup a GridFTP connection to [hostname:port]...
     * 2) transfer [remoteFileName] to local storage
     * 3) parse local file and filter books by [topic]
     */
  }
  ...
}

public class BookLister{
  BookSource source = null;

  public void setSource(BookSource source){
    this.source = source;
  }
  public BookSource getSource(){
    return source;
  }
  public void displayBooks(String topic){
    List results = getSource().getBooks(topic);
    // Iterate and display results
  }
}
```

We also must indicate the DI container to use the `GridFTPBookSource` class when injecting a

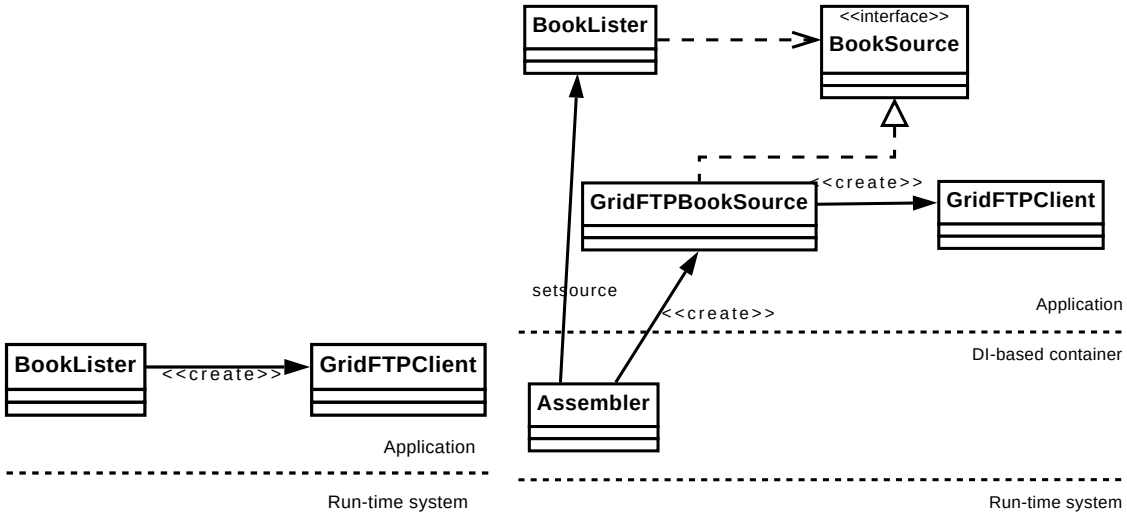
value to the `source` field. This is supported in most containers by configuring a separate file (usually in XML format), which specifies a concrete implementation and configuration information for all the components of an application, along with the dependencies that exist between these components. In the rest of the chapter, the examples will be based on Spring (Walls and Breidenbach, 2005; Johnson, 2005), a popular Java-based DI container for developing component-based distributed applications, which provides the means for Grid service injection in JGRIM. The configuration file for the example being discussed is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<components>
  <component id="myLister" class="BookLister">
    <dependency name="Source">mySource</dependency>
  </component>
  <component id="mySource" class="GridFTPBookSource">
    <property name="hostname">gridftp.books.com</property>
    <property name="port">2811</property>
    <property name="remoteFileName">remote-books.info</property>
 </component>
</components>
```

Figure 4.3 shows the class diagrams corresponding to the two versions of our book listing application. In the non-DI version shown on the left, `BookLister` directly interacts with the GridFTP client. In this way, the application logic is mixed with code for creating and configuring GridFTP, thus resulting in a poor solution in terms of flexibility and maintainability. On the contrary, in the DI version shown on the right, the implementation code for dealing with both the configuration and creation of a GridFTP connection is partially replaced by configuration information placed on a separate file, which is processed at runtime by an assembling element supplied by the DI container.

DI is an effective way to achieve loose coupling between application components. The pattern results in highly decoupled components, since the glue code for linking them together is not explicitly declared in the application code (Johnson, 2005). In the context of this thesis, a crucial implication of this separation is that the code linking components can be completely managed at the middleware level to transparently add Grid behavior to ordinary applications. For example, Grid concerns such as service discovery, service invocation and parallelization are supported by the JGRIM middleware by means of injection of built-in Spring-based m-components. Indeed, the main idea of JGRIM is to inject all Grid related services into any Java application structured as objects whose data and behavior are accessed by using the standard get/set conventions.

The next section describes the gridification process of GRATIS in the context of JGRIM.



(a) Without using DI (b) Using DI

Figure 4.3: Class diagrams for the book listing application

4.3 Gridifying Applications with JGRIM

JGRIM is a middleware for easy gridification of Java applications that materializes GRATIS concepts. Following the GRATIS philosophy, JGRIM provides a mean for developers to gridify their applications by minimizing the requirement of source code modification. In addition, JGRIM lets programmers to easily control the granularity of application components to address efficiency issues during the gridification process. JGRIM promotes separation of concerns between application logic and Grid behavior. Therefore, it conceives *gridification* as the process of shaping the source code of an ordinary application according to few coding conventions, and then non-intrusively adding Grid concerns in the form of m-components into it.

JGRIM let developers to focus first on implementing, testing and optimizing the functional code of applications, and then to Grid-enable it. JGRIM imposes little code modifications on compilation units. In addition, its programming model shares many similarities with widely used models for Java programming such as JavaBeans. Besides, the fact that Java is itself a very popular language for application development, makes JGRIM a gridification tool that can potentially benefit other kind of Java applications provided they are adapted to component-based applications.

JGRIM is materialized as an implementation of Generalized Reactive Intelligent Mobility (GRIM) (Chapter 5), a generic agent programming and execution model which allows developers to easily manage mobility of agents and resources in distributed environments. Grid applications in JGRIM (i.e. a gridified application) are application-level mobile services called Mobile Grid Services (MGS) that interact with other MGSs and use middleware-level Grid services for mobility, parallelization, service brokering and invocation. An MGS comprises the logic (what the service does), and the Grid-dependent behavior, which is configured separately from the logic and glued to the MGS at deployment time. Basically, the gridification process of JGRIM works

by semi-automatically transforming ordinary applications to MGSs.

4.3.1 Mobile Grid Services

Mobile agent technology is a well-known alternative for developing distributed applications. A mobile agent is a computer program that can migrate within a network to perform tasks or locally interact with resources (Tripathi et al., 2002). Mobile agents have some nice properties that make them suitable for exploiting the potential of Grid environments, because they add *mobility* to the capacities of ordinary agents. Some of the most significant advantages of mobile agents are their support for disconnected operations, heterogeneous systems integration, robustness and scalability (Lange and Oshima, 1999).

A mobile agent has the capacity to migrate to the location where a resource is hosted, thus avoiding remote interactions which can significantly reduce network traffic. Consider, for example, an information retrieval service. The goal of the service is to perform a data mining analysis task over several tables of a database hosted at a site S. Based on information such as local and remote workload and available bandwidth, a mobile version of the service could decide to migrate to S in order to avoid remote interactions with the database, at the cost of potentially cheaper migration overhead. The retrieval service could then employ a mobile agent in order to interact with resources efficiently.

The service provisioning within a Grid is also well suited to be managed by mobile agents (Di Martino and Rana, 2004). Scheduling, brokering, monitoring and coordination are inherently high-level tasks that require agents' abilities such as autonomy, proactivity, mobility and negotiation. In fact, several Grid systems and applications have proposed the use of mobile agents as their underlying Grid infrastructure (Fukuda et al., 2006). Mobile agents are a good alternative for providing efficient access to computational resources and supplying the basic bricks for building service-based Grids. Specifically, the approach of having mobile agents providing Grid services is very interesting, since it permits mobile agents and services to complement each other to achieve better network usage and increased efficiency (Ishikawa et al., 2004), among other advantages. Grid middleware and its services need to be highly-scalable and interoperable, since managing today's diverse and heterogeneous infrastructure for making Grid Computing a reality is an increasing challenge. Mobile agents provide a satisfactory solution to tackle down both of these problems (Lange and Oshima, 1999).

Using mobility within the Grid is not a new idea. For example, mobile agents have been successfully employed for job submission and management (Barbosa and Goldman, 2004; Fukuda et al., 2006), resource sharing (Suna et al., 2004), and resource discovery (Chunlin and Layuan, 2003; Aversa et al., 2004). In addition, Grid infrastructures such as Cactus (Allen et al., 2001), Condor (Thain et al., 2003), GridWay (Huedo et al., 2004) and GrADS (Vadhiyar and Dongarra, 2005) also rely on mobility for both application scheduling and execution. However, their migration frameworks use traditional process migration techniques, while JGRIM provides mobility at a higher level of abstraction by supporting migration for both Grid applications and resources.

Besides, the granularity of mobility is quite different, since our migration scheme only moves certain application objects rather than entire processes.

4.3.2 JGRIM Application Anatomy

The most important aspect of JGRIM is its *gridification process*, that is, the set of tasks users must follow to adapt their applications to run on a Grid. This process is illustrated in Figure 4.4. As we will see in the next sections, the gridification process of JGRIM is semi-automatic, as it roughly requires:

1. Modification of source code in order to obey some simple and standard object-oriented code conventions, so as to ensure that applications components are implicitly linked through get/set accessors. Since the get/set programming style is commonplace in object-oriented development, it is expected that most of the time this task will not be necessary or will require little effort.

2. User-supplied information about the interfaces of both implemented application components and external (functional) Grid services potentially needed by the application. Basically, this information indicates the hot spots for service injection within the application.

3. Assembling of the outputs of (1) and (2), and deployment of the newly created MGS on a particular Grid, which are performed automatically by JGRIM. Once deployed, an MGS becomes itself a functional Grid service that may in turn be injected into another application.

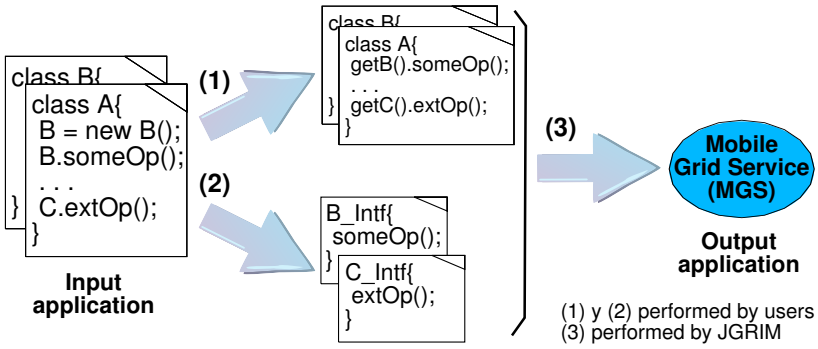


Figure 4.4: JGRIM: Gridifying applications

Ordinary applications after passing through the JGRIM gridification process become WSDL--interfaced functional Grid services with migration capabilities called MGS. Basically, an MGS is a mobile entity that is composed of two parts:

- a *stationary* part, given by a WSDL document describing the interface of the service (i.e. its operations), and a binding, which acts as a bridge between the WSDL and the mobile part of the service. As described in Chapter 2, WSDL (W3C Consortium, 2007b) is a well-known XML-based language for describing distributed services as a set of operations over ubiquitous transport protocols (HTTP, RMI, CORBA, etc.). From a WSDL specification, an MGS can determine the operations other MGSs provide, and how to interact with them. It is worth noting that the WSDL interface for an MGS is not supplied by the developer, since it is automatically built from the public methods of the MGS implementation, that is, the original application.

- a *mobile* part, consisting of a mobile agent carrying the logic that implements the service. As such, the agent can potentially move to other hosts in order to find and locally access required resources, or even other MGS. Additionally, the agent is instructed with the Grid-dependent behavior configured for the MGS by combining it to the service logic. The next section presents further details on how this is done.

  Note that, since Grid-aware applications in JGRIM are essentially mobile entities whose runtime behavior follows a particular execution model, JGRIM acts not only as an injector of mobility but also as a concrete provider of migratory capabilities. In this sense, JGRIM can be seen both as a gridification tool and a Grid middleware that provides mobility services to applications.

When an MGS is deployed on a host, its IDL information is locally installed, and its associated agent is launched. Upon launching, a proxy to the mobile agent is created in order to hide its real location. The mobile agent informs its location to its proxy after processing each operation request, in a piggybacking fashion. Moreover, each proxy maintains a mailbox where client requests are queued. When the agent finishes processing a request, the proxy picks an unserved request from the mail box and forwards it to the agent. Figure 4.5 depicts the anatomy of a running gridified application.

The runtime support for MGSs is implemented through Java servlets (Hunter and Crawford, 1998). It provides low-level services to agents such as execution, mobility, system resource monitoring (CPU load, bandwidth, memory, and so forth) and request forwarding. Mobility here is mostly concerned with marshaling agents' execution state into a network-transferable format, unmarshaling received agents and resuming their execution. This is partially supported by means of JavaFlow (Apache Software Foundation, 2006), a library from Apache for capturing the execution state of a running Java application.

The following paragraphs explain the Grid service injection support of JGRIM.

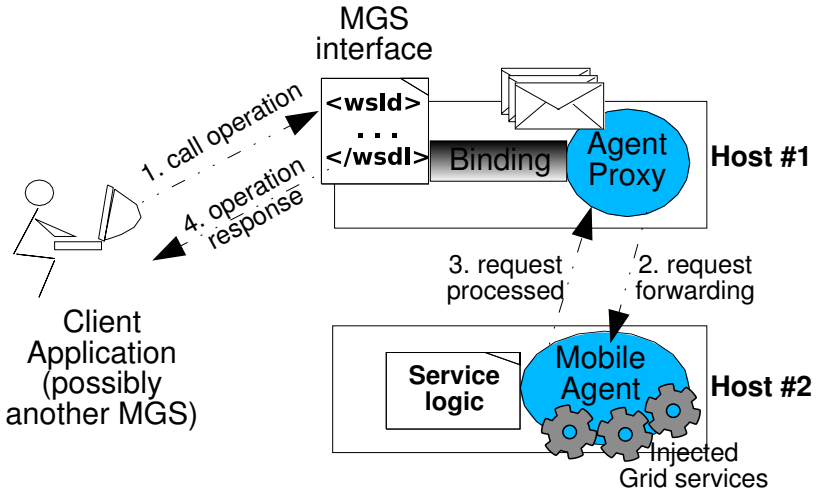


Figure 4.5: Elements of an MGS

4.3.3 Grid Service Injection

As stated at the beginning of this chapter, GRATIS conceives ordinary applications as being composed of *logic* and *dependencies*. The logic is the portion of the application implementing pure application logic, that is, code for performing calculations, interacting with data resources, and requesting services from other applications. These data resources and external services are the dependencies of an application. An individual dependency specifies, by means of a well--defined interface, the conventions needed to appropriately interact with a certain application component or functional Grid service. After gridification, individual dependencies may have configured one or more Grid metaservices.

Figure 4.6 summarizes the concepts introduced above. According to JGRIM, dependencies may also have an attached *policy*, which non-intrusively customizes the way application components or external services are accessed when executing in a Grid setting (e.g. to increase efficiency). At the middleware level, both policies and dependencies are materialized as JGRIM built-in components, or m-components. Basically, these m-components are intended to represent the Grid-related behavior of applications.

4.3.3.1 The Service Discovery/Invocation m-Component

Unless otherwise indicated, all dependencies for an application are assumed to refer to external MGSs, Web Services or any other kind of service-like entity described through a WSDL interface. The Java interface (e.g. `BookSource` in the example of Section 4.2) for every dependency of a JGRIM application is in fact transparently associated to a service discovery m-component. This kind of built-in component accepts service requests from the application upon it depends, inspects registries to find a service whose WSDL interface matches the dependency interface,

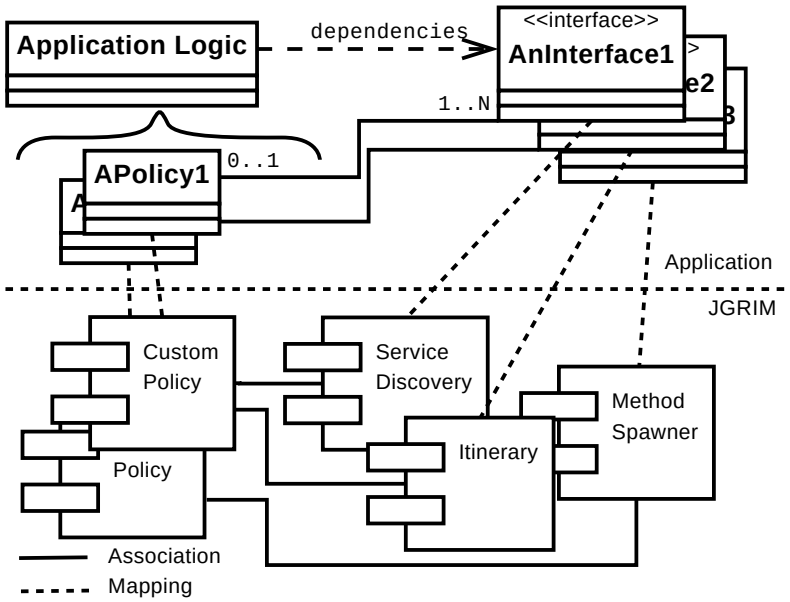


Figure 4.6: Dependencies, policies and JGRIM m-components

and invokes operations on that service[1]. In this way, the user is freed not only from the burden of searching functional Grid services, but also from dealing with low-level details such as addresses, protocols, ports, etc. to invoke these services.

Moreover, based on the support offered by this m-component, developers may decide not to implement certain applications components but relying on functional service injection capabilities, thus drastically reducing implementation effort. Note that this type of reuse is precisely the cornerstone of the SOA paradigm.

Figure 4.7 shows a diagram illustrating how JGRIM isolates applications from discovering and invoking functional Grid services. `ServiceDiscoverer` materializes the service discovery m-component previously mentioned. Instances of this m-component, which are automatically injected into the application, act as proxies to functional Grid services by converting any incoming method call to the corresponding WSDL operation request. For each dependency –and therefore for each interface– a proxy is injected, which is in charge of discovering and then accessing any of the available Grid services offering the operations declared in the corresponding interface. In this way, methods calls issued at the application level on a particular dependency are transparently intercepted by the associated proxy and forwarded to a concrete service instance within the Grid.

Conceptually, a service discoverer can be viewed as a simple service broker that, given a specific list of operation signatures, is capable of finding concrete Grid services whose exposed interface is a superset of the input list. In addition, the service discoverer is responsible for transforming

[1]According to the WSDL specification, a Web Service may be composed of one or more operations

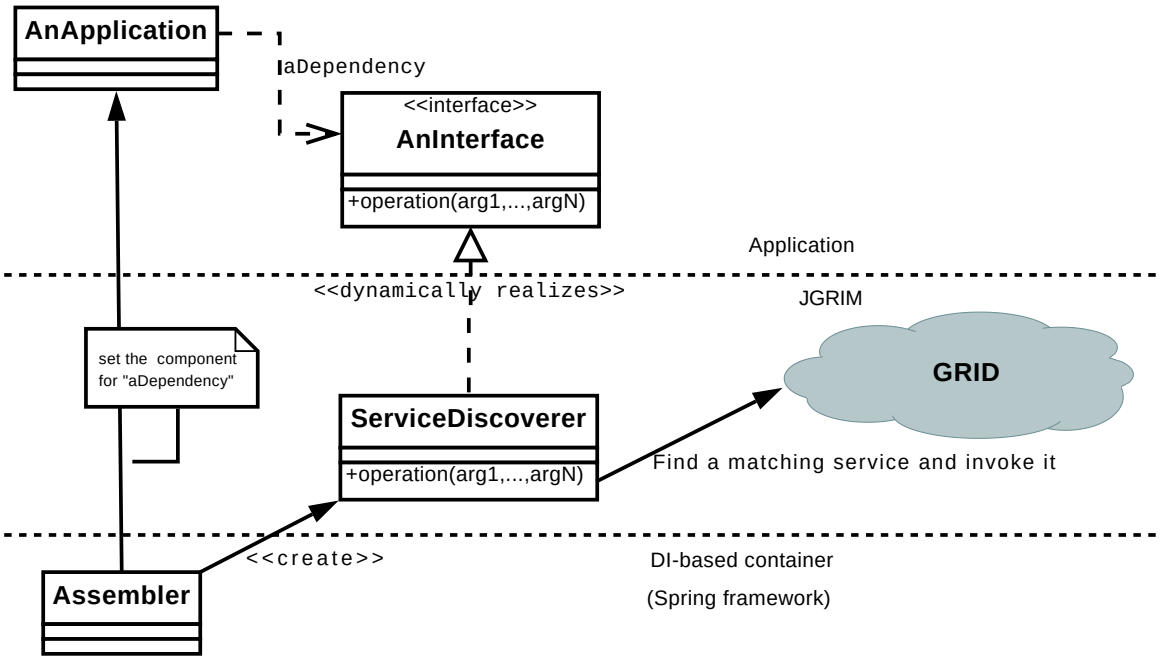


Figure 4.7: Dependencies and service discovery/invocation m-components in JGRIM

a call on any of these operations to an operation request on any of the service instances that were initially found. Note that the materialization of the discovery mechanism may involve, for example, the inspection of a centralized service registry or the use of Peer to Peer (P2P)protocols. In fact, the current implementation of this mechanism in JGRIM (see Section 6.2.1) is based on UDDI registries.

4.3.3.2 The Policy m-Component

Another aspect that is represented by means of m-components is *policy management*. Basically, policies represent a mechanism that allows developers to express, separately from the application logic, customized reconfiguration strategies for mobile applications in order to achieve better performance (Montanari et al., 2004). Particularly, JGRIM provides a policy-inspired application tuning support that let programmers to customize the way applications interact with Grid services without altering application code.

JGRIM applications use policies mainly for managing MGS mobility and resource access. For example, one could instruct an MGS to move to a powerful host every time a specific CPU--intensive operation of a certain application component is executed, or specify filtering actions on the available providers for an external service component. Individual policies can be attached to the operations of one or more dependencies. Basically, a JGRIM policy is a plain Java class that allow developers to specify:

- Customized pre and post actions that are executed by JGRIM before and after an invocation to any of the operations of the associated dependency takes place. This is useful to

keep the "conversation" between an individual application component and its dependencies, which in turn may serve as a mean of improving efficiency (e.g. caching invocations results), performing debugging (e.g. diagnosing communication errors with external services) or just carrying out profiling tasks.

- Code to dynamically select the particular source for an external component and the way it should be accessed. Within a Grid environment, many providers for the same service interface are usually available, thus applications may want to decide which instance should be used. On the other hand, an individual service instance may be contacted in several ways, such as moving the requesting MGS to the service's location or remotely calling it. These decisions can be specified only for dependencies to functional Grid services accessed through discovery/invocation or itinerary m-components.

Dependency type (target component)	**Pre/Post actions**	**Service instance/ access method selection**
Service (external component)	√	√
Service (internal component)	√	–
Self dependency (this)	√	–

Table 4.1: Aspects that can be controlled through policies, according to different dependency types

Table 4.1 summarizes the aspects that can be customized by a policy depending on the type of the dependency to which the policy is associated. After gridification, JGRIM maps all policy declarations to a special middleware-level m-component, thus service discovery/invocation m-components can reference and use these policies. In addition, dependencies can be configured to employ policies in order to establish, for example, a custom ordering for accessing services when using itineraries. The rationale behind this customization mechanism is the GRIM model, which will be described in the next chapter. In GRIM terminology, JGRIM policies are conceptually defined as *agent-level policies*.

As explained in past sections, upon gridification, the application logic is mapped to an MGS. This is done by automatically modifying the application source in order to inherit from specific classes of the JGRIM API that provide basic high-level primitives for handling agent mobility and managing agent state information. Similarly, interfaces for dependencies and classes implementing custom policies are mapped to m-components, and combined into a single configuration file which is then wired to the MGS functional behavior. According to GRIM, those interfaces are known as the *protocols*[2] of the MGS.

[2]The term should be understood here in the context of object-oriented programming

The next section presents a comprehensive example in order to clarify the ideas exposed so far.

4.3.3.3 The Parallelization m-Component

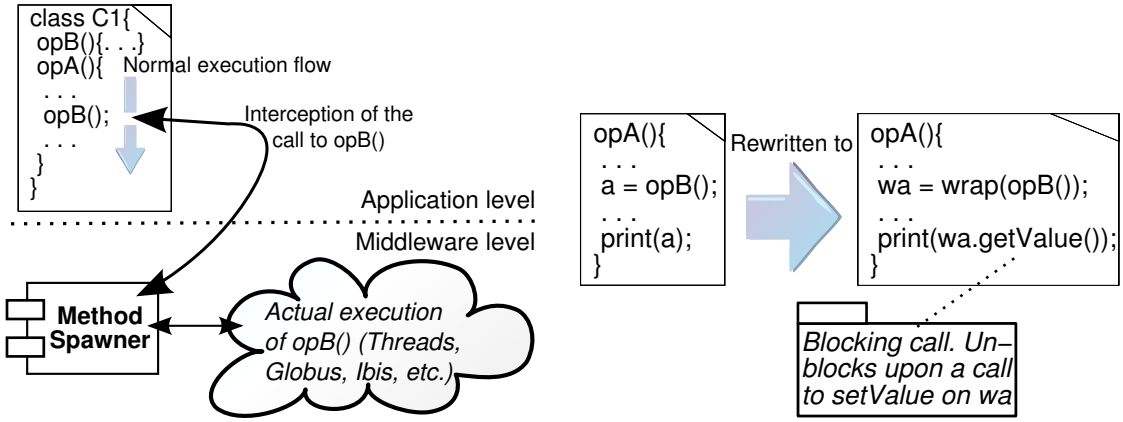
A fundamental service of any Grid is parallelization. Certainly, simultaneously executing the same task or a set of closely-related tasks can dramatically improve the performance of Grid applications. Indeed, most Java-based platforms proposed for Grid programming provide some form of execution parallelization, mainly in the form of *method spawning*. Basically, the idea is to distribute the execution of certain methods to atomically run them on different hosts. For example, Ibis (Chapter 3) let programmers to declare a special marker interface whose methods corresponds to those operations within a class which should be spawned. Another example of a Java-based language for parallel programming that uses method spawning techniques is JCilk (Danaher et al., 2006).

From the point of view of the JGRIM programming model, an interface similar to an Ibis marker is considered as a dependency that do not refer to external Grid services, but to operations provided by the application. In other words, an application offering services usually depends upon itself, since these services internally use each other. For example, a component C_1 might provide two operations A and B, with A depending on several independent calls to B. Overall, C_1 has a dependency to itself, whose interface includes an operation with exactly the same signature as B.

In this sense, JGRIM exploit this notion to transparently inject parallelization: developers can specify *self-dependencies* that are automatically mapped to a `MethodSpawner` built-in component (Figure 4.8 (a)). The `MethodSpawner` transparently parallelizes the invocation of any method declared within those dependencies by creating different child tasks that can be sent to remote hosts to improve performance. Section 4.3.4 exemplifies the usage of this m-component.

Task execution within a `MethodSpawner` can be managed in several ways. A very interesting scheme is to delegate the execution of these tasks to existing job submission services such as those offered by Ibis or Globus's GRAM, which are suited to handle the execution of CPU-intensive applications. In addition, reusing these services allows applications to indirectly benefit from useful features like load balancing, fault tolerance and execution monitoring. Currently, implementation of the JGRIM `MethodSpawner` is based on the execution and parallelization services provided by the Ibis platform. In the next chapters, a detailed explanation and evaluation of this support is presented.

JGRIM complements the above mechanism by transforming the application source code so as to wrap the result of a (spawnable) method call with a special object, and replace any further reference to that result with an invocation to a blocking `getValue` method on that object (Figure 4.8 (b)). The execution of the method is actually handled by a `MethodSpawner` in a



(a) The Method-Spawner built-in component (b) Sincronization of a spawned computation

Figure 4.8: The Parallelization m-Component

separate thread, which is in charge of contacting an external Grid execution service (in this case Ibis) and then calling the corresponding `setValue` method on the wrapper object once the results are available. In this way, both the task of spawning methods and waiting for their results are totally transparent to the application programmer. Another very important benefit of this separation is that applications can be configured to use different job submission services without requiring source code modifications.

4.3.3.4 The Itinerary m-Component

JGRIM supports the notion of *itinerary*. In most mobile agent-based platforms, an itinerary is an ordered list of sites used by agents to locally perform a single task or a set of tasks on these sites, one at a time. JGRIM provides a m-component which, upon creation, locates the external functional services adhering the interface of the dependency to which an application component is associated, and store them into a list. The m-component delegates an incoming service invocation to the current item of the list by moving the MGS to the service location. When the list reaches its end, an exception to the MGS is thrown. Besides, the itinerary is incrementally updated as new services are discovered.

JGRIM itineraries are useful when applications need to contact more than one provider for the same service functionality. For example, a temperature forecaster could be easily gridified by declaring an itinerary-like dependency (`traveler`) that searches for Grid services offering a `predict(latitude, longitude, day)` operation:

```
public interface SimpleForecasterInterface{
  public long predict(long latitude, long longitude, Date day);
}

public class ProbabilisticForecaster{
  public static final int PROVIDERS = 5;
```

```
  // An itinerary-like dependency
  SimpleForecasterInterface traveler;

  public void setTraveler(SimpleForecasterInterface traveler){
    this.traveler = traveler;
  }
  public SimpleForecasterInterface getTraveler(){
    return traveler;
  }
  public long forecast(long latitude, long longitude, Date day){
    long[] predictions = new long[PROVIDERS];
    for (int i=0; i<PROVIDERS; i++){
      predictions[i] = getTraveler().predict(latitude,longitude,day);
    }
    // Probabilistic forecasting
    return forecast(predictions);
  }
  protected long forecast(long[] forecasts){...}
}
```

Each method call served by the m-component results in a request to a different service provider. After a number of results has been obtained, the application might apply probabilistic methods in order to return a more accurate prediction of the temperature.

Note that an itinerary m-component can be viewed as a special type of service discovery/-invocation m-component. They behave almost the same, except that the former has some notion of state. While the latter delegates every single method call coming from the application to any matching service it discovers, itinerary m-components contact all those services which has been initially discovered when the m-component was injected.

4.3.4 An Example: The k-Nearest Neighbor Classifier

From the programmer's perspective, there are a few conventions to keep in mind before using JGRIM. First, all dependencies to services must be labeled with an identifier, and a type (service, itinerary or self) to provide information to JGRIM about how to map these dependencies to built-in components. Second, any reference to a dependency within the application code must be done by calling a fictitious method *getXX*, where *XX* is the identifier given to that dependency, instead of accessing the dependency directly (XX.operation()). For example, if an application reads data from a file, instead of accessing it as dataFile.read(), it should be accessed as getDataFile().read(). JGRIM then automatically modifies the source code so as to include the necessary instance variables and getters/setters methods. After that, the compiled

version of the code is instrumented to enable the application for performing method spawning and mobility.

This section shows the gridification process for an existing implementation of k-nearest neighbor algorithm (k-NN). k-NN (Dasarathy, 1991) is a popular supervised learning technique used in data mining, pattern recognition and image processing. The algorithm is computationally intensive, hence it is a suitable application to be deployed on a Grid environment.

k-NN classifies objects (also called instances) by placing them at a single point of a multidimensional feature space. The training examples (also called dataset) are mapped into this space before instance classification, thus the space is partitioned into regions according to class labels of the training samples. A point in the space is assigned to the class C if it is the most frequent class label among the k nearest training samples.

Suppose we already have a Java application implementing this algorithm and we want to gridify it with JGRIM. Roughly, the application consists of a number of helper classes and a `KNN` class declaring three operations:

- `classifyInstance`, which computes the class label associated to a particular instance,
- `classifyInstances`, which is analogous to `classifyInstance` but operates on a list of instances, and
- `sameClass`, which tests whether two different instances have associated the same class label.

Among these helper classes is the component that provides the means to access the dataset based on the specific storage mechanism (e.g. a file), which is in turn called by the above methods to perform classification of new instances. Basically, the structure of the application code could be as follows:

```
public class KNN{
  protected int k;
  FileDatasetReader dataset;

  KNN(int k){
    this.k = k;
    this.dataset = new FileDatasetReader();
  }
  public double classifyInstance(Instance instance) {. . .}
  public double[] classifyInstances(Instance[] instances) {. . .}
  public boolean sameClass(Instance instA, Instance instB) {. . .}
}
```

```
public class FileDatasetReader{
  public Instance[] readItems(int start, int end) {. . .}
  public int size() {. . .}
  public int dimensions() {. . .}
}
```

As explained previously, gridifying an application involves to take its code, along with some user-provided information, and generate the corresponding MGS. Basically, the user must define the class within the application he wants to expose as an MGS (in this case, the KNN class), and provide a list of pairs *[identifier, JavaInterface]* specifying all the existing dependencies[3]. Optionally, a third argument representing the dependency type can be specified, and a fourth configuring a custom policy.

Apart from the computational resources itself, k-NN needs a data resource with the dataset in order to classify a single instance. In JGRIM, this is modeled through a dependency to an external component. We expect that data resource to expose an interface for reading items and metadata (size and number of dimensions) from the dataset, so we provide a Java interface for this matter (`DatasetInterface`), and a dependency with this interface identified as *dataset* is declared in the application. The gridified version of the code is[4]:

```
public interface DatasetInterface{
  public Instance[] readItems(int start, int end);
  public int size();
  public int dimensions();
}

public class KNN extends jgrim.core.JGRIMAgent{
  protected int k;
  DatasetInterface dataset;

  KNN(int k){
    this.k = k;
  }
  public void setdataset(DatasetInterface dataset){
    this.dataset = dataset;
  }
  public DatasetInterface getdataset(){
    return dataset;
  }
```

[3]The development of a plug-in for the Eclipse SDK to help programmers define dependencies and create/configure policies is underway.

[4]Some of the class names shown in the examples of this chapter may differ from the actual JGRIM API discussed in Chapter 6.

```
  public double classifyInstance(Instance instance) {. . .}
  public double[] classifyInstances(Instance[] instances) {. . .}
  public boolean sameClass(Instance instA, Instance instB) {. . .}
}
```

and the automatically generated configuration file is:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="jgrim_KNN" class="KNN">
    <property name="dataset">
      <ref local="jgrim_dataset"/>
    </property>
  </bean>
  <!-- A service discovery m-component -->
  <bean id="jgrim_dataset" class="jgrim.core.ServiceDiscoverer">
    <property name="expectedInterface">
      <value>DatasetInterface</value>
    </property>
  </bean>
</beans>
```

The application declares a dependency to a data resource named *dataset*, which has been previously modeled through the `DatasetInterface` interface. Besides, proper getter/setters methods for interacting with that dependency were automatically added. As the reader can see, the resulting code is very clean, since it was not necessary to use any particular class from the JGRIM platform during the gridification process. The only requirement for the programmer was to add calls to a (not yet existent at implementation time) `getdataset` method at the places where the dataset is accessed. Furthermore, it is very easy to change the real source of the dataset for testing purposes, by simply replacing the *jgrim_dataset* component in the configuration file. A quantitative report on the gridification effort for the k-NN algorithm, from the programmer's point of view, can be found in Chapter 7.

4.3.4.1 Parallelization

A natural way to implement the `sameClass` operation is by issuing two different calls to `classifyInstance` and then simply compare the results, thus improving code reusability. These calls are inherently independent between each other, hence they are computations suitable to be executed concurrently.

Let us exploit this fact by injecting parallelization into the `sameClass` operation. Basically, the idea is to declare a self-dependency to those operations whose execution should be parallelized. Let us identify this dependency as *self*, and specify its corresponding Java interface:

```
public interface SpawnInterface{
  public double classifyInstance(Instance instance);
}
```

As a consequence, a new component is added to the configuration file previously shown:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="jgrim_KNN" class="KNN">
    . . .
    <property name="spawner">
      <ref local="jgrim_spawner"/>
    </property>
  </bean>
  . . .
  <!-- A method spawning component -->
  <bean id="jgrim_spawner" class="jgrim.core.MethodSpawner">
    <property name="expectedInterface">
      <value>SpawnInterface</value>
    </property>
  </bean>
</beans>
```

In order to adequately interact with the *self* dependency in the `sameClass` operation, the programmer must replace the two implicit calls to `classifyInstance` through the *this* reference variable by calls to a fictitious method `getself`. Similarly to Ibis, the only extra programming convention needed for the spawning technique to work is that the results of the spawned computations must be placed on two different Java variables. Further references to any of these results will block the execution of the `sameClass` operation until they are computed by the `MethodSpawner` component. A sequence diagram showing the interaction between the objects involved in the computation of the `sameClass` operation is depicted in figure 4.9.

4.3.4.2 Policy Management

To illustrate the runtime behavior of our gridified classifier service, let us suppose a scenario consisting of several JGRIM-enabled sites where some of them have a copy of the dataset stored

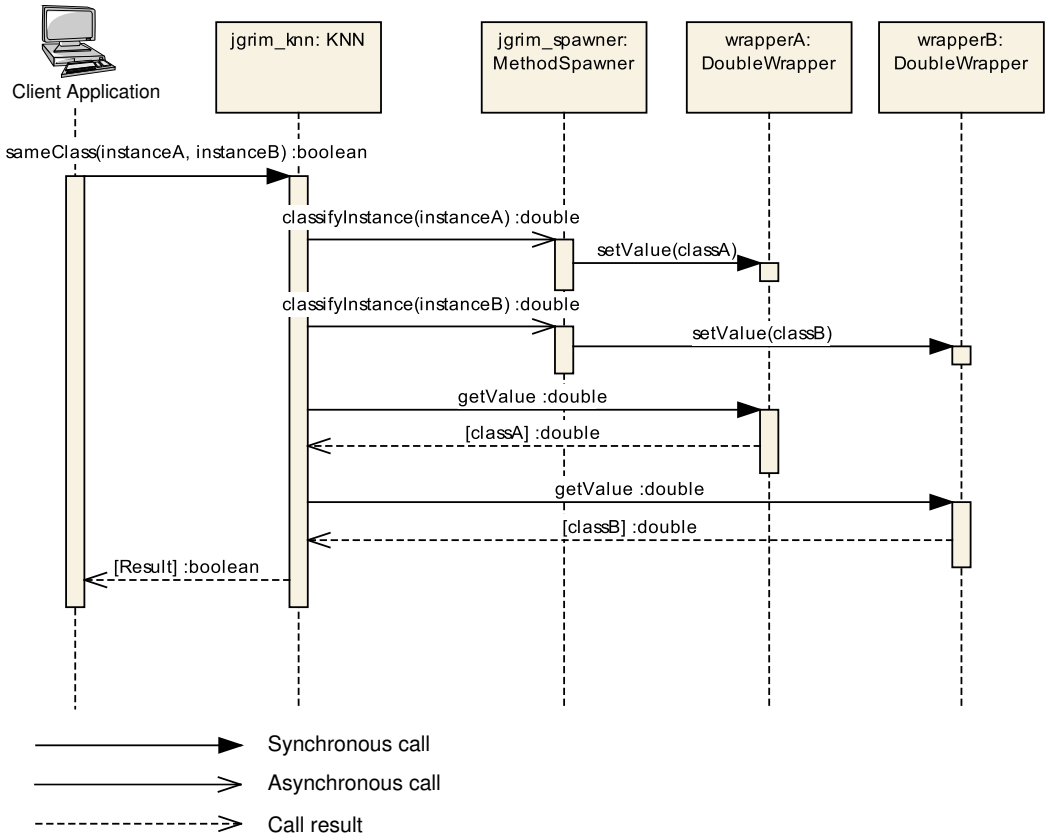


Figure 4.9: Parallelization of the `sameClass` operation: sequence diagram

on a database. All occurrences of the dataset has been previously wrapped with a single Web Service, providing the operations needed by the MGS for querying the dataset. Finally, the latency between any pair of sites could vary along time. The same applies to CPU load at any site.

Since our MGS works by reading blocks of data from the dataset and then performing a compute-intensive computation on them, availability of both network bandwidth and CPU should be seriously taken into account. Accessing a service from a site to which the MGS current location experience a high latency might increase response time. Moreover, processing a block of data on a loaded site might lead to inefficient use of computing resources. We could overcome this situation by attaching a policy to the *dataset* dependency. To code a policy, the programmer must specify his access decisions by implementing four separate methods of the `PolicyAdapter` class:

```
import jgrim.core.Constants;
import jgrim.core.policy.PolicyAdapter;
import jgrim.metrics.Profiler;

public class DatasetPolicy extends PolicyAdapter{
  public String accessWith(String methodA, String methodB){
    return Constants.INVOKE;
```

```
  }
  public String accessFrom(String siteA, String siteB){
    Profiler pr = Profiler.getInstance();
    double latA = pr.profile("latency", pr.getLocalHost(), siteA);
    double latB = pr.profile("latency", pr.getLocalHost(), siteB);
    return (latA < latB) ? siteA : siteB;
  }
  public void executeBefore(){
    String site = Profiler.getInstance().leastLoadedSite();
    ((JGRIMAgent)getOwner()).move(site);
  }
  public void executeAfter(){. . .}
}
```

For simplicity reasons, exception handling is omitted. The previous code works as follows: every time the MGS calls the dataset service, `DatasetPolicy` is evaluated. Firstly, the method `executeBefore` causes the MGS to migrate to the least loaded host. From there, the policy instructs our MGS (through methods `accessWith` and `accessFrom`) to remotely invoke the service which is hosted at the site that offers the lowest latency. After accessing the service, the code placed within `executeAfter` method is executed. For now, it should be clear that it is very easy to tune JGRIM applications by means of policies, which customize the way Grid services are accessed without altering the MGS behavior. In the next chapter, a more deep discussion on the policy mechanism is presented.

It is worth noting that `DatasetPolicy` will be activated upon invocations on any operation from the `DatasetInterface` interface, because by default a policy handles the execution of any method within the interface of the dependency to which the policy is attached. However, operations `size` and `dimensions` are very lightweight in terms of bandwidth consumption, and they are invoked only once during the classification process, thus policies are not needed for them. But, two unnecessary and potentially expensive MGS migrations are triggered anyway. In this sense, the policy mechanism also let programmers to attach policies to single operations rather than to entire interfaces.

By adding a simple policy, the MGS is now able to make mobility decisions according to availability of both CPU and network resources across sites. The weak point of the approach, as shown in the code, is that it is necessary to know details of a small subset of the JGRIM API in order to code a policy. As pointed out in the previous chapter, this is a problem suffered by many of the tools proposed for Grid development. However, the approach presented so far do not come into contradiction for two reasons. First, API details are circumscribed only to policy coding, as they are never present in the service logic. Second, the usage of policies for gridifying applications is not mandatory, since decisions regarding MGS mobility and service invocation can be delegated to the JGRIM middleware, at the potential cost of decreased performance.

4.4 Conclusions

This chapter described GRATIS, a new approach to gridification that is essentially based upon the concept of Dependency Injection. In DI, explicit references between application components are removed from the source code and transparently injected by a runtime system instead. GRATIS takes advantage of this idea to dynamically inject Grid services to ordinary applications without forcing developers to modify the application logic. A fundamental aspect of GRATIS is that it sits on top of existing Grid middlewares and infrastructures, therefore leveraging existing Grid services and resources. Lastly, GRATIS is strongly inspired on SOA concepts, thus it promotes reusability of both Grid services and user applications.

The chapter explained the approach in the context of JGRIM, a middleware that supports the GRATIS gridification method for component-based Java applications. JGRIM lets developers to easily code and deploy Grid applications while keeping them away from most Grid-related details. Besides providing mechanisms for porting applications to the Grid with little effort, JGRIM aims at addressing performance issues through the mechanisms prescribed by a mobile agent-based, easy-to-tune application execution support called GRIM.

Basically, GRIM models common interactions between mobile applications and resources in distributed systems. The next chapter describes this model.

Chapter 5

GRIM

GRIM (Mateos et al., 2005; Mateos et al., 2007b; Mateos et al., 2007c) is a generic, programming language-neutral reference model for building mobile agent-based applications. GRIM prescribes an execution model for mobile agents that is based upon the concept of *binding*, that is, a middleware-level mechanism for transparently and efficiently supplying agents with available instances of requested resources. In essence, GRIM promotes separation of concerns between application logic and non-functional aspects of mobile applications related to distributed resource access, agent mobility and application tuning.

As mentioned in the previous chapter, gridified applications in JGRIM are mobile agents called MGS whose runtime behavior is based on the GRIM model. As such, an agent is basically composed of application logic and some external configuration (i.e. non-functional concerns) that dictates how the agent behaves with respect to mobility management and resource access when executing on a distributed system.

The rest of the chapter is organized as follows. The next section introduces GRIM and discusses its related concepts. Then, Section 5.2 explains the internals of the execution model prescribed by GRIM for agent-resource interaction. Finally, Section 5.3 briefly summarizes the contents of the chapter.

5.1 Overview of GRIM

GRIM is a generic agent execution model that abstracts away common interactions between mobile agents and resources when building mobility-based distributed applications. GRIM generalizes Reactive Mobility by Failure (RMF)[1] (Zunino et al., 2005b), a transparent migration mechanism that aims at reducing the effort of developing mobile agents by automating some decisions about mobility. In RMF, a *failure* is defined as the impossibility of an executing mobile agent to find some needed resource (e.g. data, services, etc.) at the site at which the agent is

[1]For a comprehensive discussion on the RMF mechanism, see (Zunino, 2003)

running. A single failure is handled by RMF by moving the failing mobile agent to a remote site containing instances of the requested resource. Basically, GRIM takes this idea further by enlarging the set of methods that are employed to handle failures, and providing flexible decision mechanisms so that developers can tailor GRIM according to specific application requirements.

GRIM is built upon the concept of *binding*, which is conceived as the process of matching an agent resource request to the actual resource instance able to serve that request. Generally speaking, requests are composed of resource interface information, which specifies the desired interaction "contract" with resources, and metadata information, which defines concrete values for the various properties that identify individual resource instances matching the same interface information. Metadada information is useful to specify constraints over the size, availability, owner, etc. of a resource. For example, an agent might be interested in contacting a service (i.e. the resource) for performing a keyword-based search (i.e. the interface information) whose provider is "Amazon".

Figure 5.1 depicts an overview of the GRIM model. The execution units prescribed by GRIM are mobile agents composed of two parts: *behavior* and *protocols*. Behavior is the stationary implementation of an agent, that is, the code not concerned with mobility. Protocols represent resources potentially needed by the agent along its lifetime. The execution environment for agents residing at a physical site is called a *host*, which is introduced in the next subsection. Hosts are responsible for hosting agents as well as resources. The terms "agent" and "mobile agent" will be used interchangeably throughout the rest of the chapter.

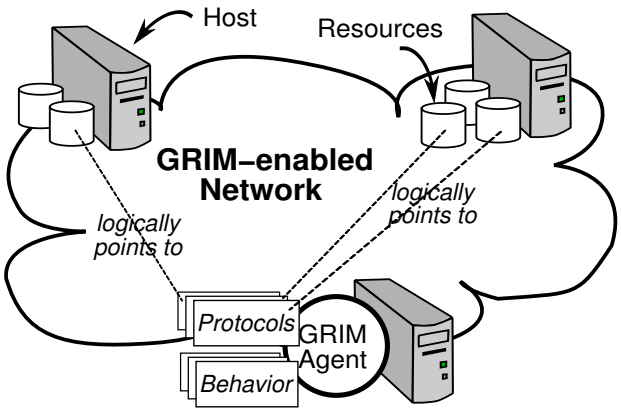


Figure 5.1: The GRIM model: core concepts

As mentioned above, an agent behavior corresponds to the logic implementing that agent. This logic usually makes use of resources external to the agent which are described by means of protocols. A protocol is essentially the interface –methods and conventions– exposed by a resource to which the agent agrees in order to access the resource. Indirectly, protocols indicate the points within the agent's behavior which potentially may need a binding. For example,

the behavior of an agent looking for a phrase within a file would be that of implementing a string matching algorithm over the file contents. Here, a protocol is required to indicate the algorithm needs an external resource (i.e. the file), and the expected interface for manipulating that resource (e.g. open, read and close methods). In addition, metadata such as name, size and permissions might be associated to the protocol, thus an individual file is identified. It is up to the middleware to bind each operation issued on the file resource with the code implementing the actual interaction. For instance, in the Grid context this may be a Web Service providing operations for accessing files or a wrapper to a Grid file system.

GRIM states that, upon any attempt to access a resource (e.g. the file in the above example), a *middleware trap* is generated. As a consequence, the trap is reactively handled, that is, GRIM automatically do whatever is necessary in order to obtain a resource whose exposed interface adheres to the protocol causing the trap. For instance, GRIM may move the agent to a site offering the required resource, or remotely invoke the resource instead. Section 5.2.1.1 discusses the mechanisms provided by GRIM to handle traps.

Moreover, not all requests to external resources trigger traps, but only those points of an agent's behavior associated with a protocol. If we map this to JGRIM, it means not all calls to a *getXX* method within an MGS will trigger a trap. The binding process will be activated only if *XX* has been declared as a dependency to an external service. Conceptually, by declaring protocols the programmer is able to select which points of an agent code may need to be bound to resources and services. At an extreme, an agent may not declare any protocol. This implies that the programmer is in charge of manually finding and accessing resources, and even performing agent mobility. However, note that this is not suitable when building applications for complex and distributed execution environments like the Grid, where manually handling resource access and application execution usually involve difficult, laborious and error-prone tasks such as service discovery, invocation, failure handling, parallelization, and so forth.

5.2 GRIM runtime support

The execution model of GRIM is able to automate decisions on when, how and what site within a network to contact to satisfy agents resource needs. GRIM is based on the notion of reactive mobility (Fuggetta et al., 1998), and as such it is founded on the idea that an entity external to agents helps them to handle traps. Those entities are stationary (i.e. non-mobile) agents called Protocol Name Servers (PNS) agents. In contrast, in schemes based on proactive mobility (Fuggetta et al., 1998), migration is exclusively managed within agents, which means programmers must explicitly provide code for handling mobility when building applications. Programming toolkits following this philosophy include at least a "move" or "go" API primitive to manually move an agent from one location to another. Proactive mobility is often referred as "explicit mobility", whereas reactive mobility is also known as "implicit mobility".

The runtime platform residing at each physical site capable of hosting and providing support for executing GRIM agents is called a *host*. Basically, hosts provide resources such as databases,

libraries or services to agents. A set of hosts such as all of them know one another conform a *logical network*. A logical network is a GRIM-enabled network that groups hosts belonging to the same application or some closely related applications. Figure 5.2 illustrates this concept. As shown in the figure, two logical networks (A and B) have been configured. Network A is composed of hosts H_1, H_2 y H_3, while network B links hosts H_2 and H_3. As the reader can observe, a single host may be part of more than one logical network. Consequently, resources owned by the host can potentially be accessed by applications belonging to any of those logical networks.

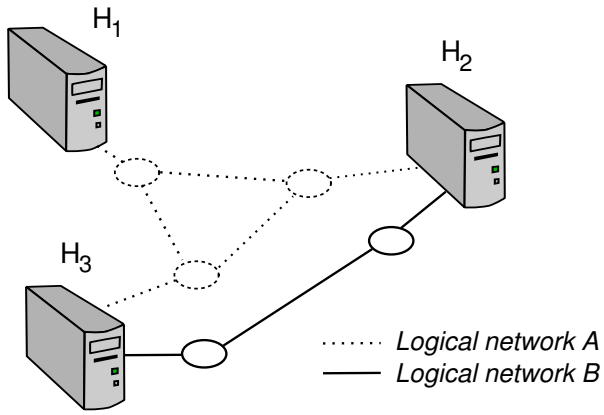


Figure 5.2: Logical networks in GRIM

Each host contains one or more PNS agents. PNS agents are responsible for managing information about protocols offered at each host, and for searching for the list of resource instances matching a given protocol whenever a trap occurs. Just like the UDDI machinery or the Globus MDS-2 metaservice serve as a vehicle for service discovery in Grid applications, PNS agents are intended to provide brokering facilities for resource discovery in mobile applications.

A host offering resources registers on its local PNSs the protocols and metadata information associated with these resources. Then, PNS agents announce this new information to its peers in the logical network(s). Figure 5.3 shows the relationships between agents, a host and its associated PNSs.

It is worth pointing out that GRIM does not prescribe any particular mechanism for dealing with protocol information at the PNS level. Both discovery and announcement of resources could be materialized with either a registry-based publication scheme, multicast and Peer to Peer technologies, or even hybrid approaches. For example, WS-Log (Mateos et al., 2007b), a platform for Web programming based on GRIM and Prolog, combines a multicast-based communication facility named GMAC (Gotthelf et al., 2005) and UDDI registries (OASIS Consortium, 2004)

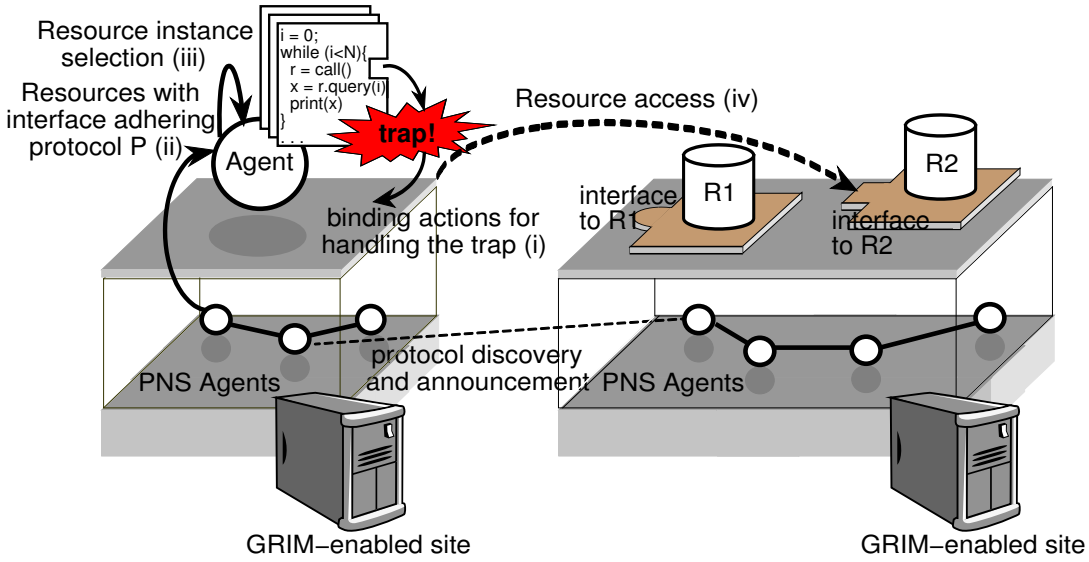


Figure 5.3: Overview of the execution model of GRIM

for managing protocol information. As for JGRIM, protocols are managed similarly to WS-Log. Other alternatives for protocol management include, for example, the materialization through the use of Semantic Web technologies (Berners-Lee, 1999). An incipient work in this line is Apollo (Mateos et al., 2006b; Mateos et al., 2006a), an infrastructure that extends the WS-Log platform with semantic-based service matching and discovery capabilities. Apollo enhances all protocols describing Web Services by annotating their operations with concepts extracted from shared ontologies.

5.2.1 Resource binding

When a middleware trap occurs, there may be many hosts offering the required resource. For example, a database could be replicated across several hosts, a file could be hosted at more than one site, or the same service could be offered by many different providers. GRIM is able to autonomously decide which host to contact in order to access the requested resource. In addition, since depending on the nature of a resource several access methods may be suitable, GRIM can apply different tactics to select the most convenient one. Both, the tasks of contacting a host and choosing an access method are decided by GRIM through *policies*. Policies are decision mechanisms based on platform-level metrics such as CPU load, network traffic, distance between sites, agent size, and so on. For example, one may specify that any access to a large remote database should be done by moving the requester where the database is located, rather than performing a time and bandwidth-consuming copy operation of the data from the remote site. Besides, GRIM lets agent developers to define custom policies to adapt the model to fit specific application requirements. In this latter case, decisions may be based on system metrics as well as application-specific information such as execution parameters, user-supplied constraints, etc.

The next section takes a deeper look at the access methods provided by GRIM for agent-resource

interaction. Then, subsections 5.2.1.2 and 5.2.1.3 discuss in detail the policy support of GRIM at both middleware and application level, respectively.

5.2.1.1 Accessing resources

Unlike its predecessor RMF (Zunino et al., 2005b), which is an agent execution model mainly designed to automate mobility decisions such as when and where to move an agent, GRIM provides mobility beyond agent code, hence GRIM "generalize" agent mobility. Note that blindly moving an agent every time some required resource is not locally available can lead to situations where performance is bad, therefore rendering agent mobility impractical (Zunino et al., 2005a). Here are two representative examples of this fact:

- When the size of an agent is greater than the size of a requested resource. Clearly, it is more convenient to transfer a copy of the resource[2] from the remote host, instead of moving the agent to that host. This approach saves network bandwidth at the cost of using more system resources on the local host (e.g. disk space for storing transferred files).
- The requested resource is a remotely-callable resource, such as a Web Service, where a transfer is usually not feasible. In particular, Web Services –apart from implementation code– have dependencies to software modules or libraries and configuration information that are either not available or do not apply in other hosts. For these reasons, the proper way to use a Web Service is by remotely invoking the operations defined by the service, thus transferring only (potentially small) input arguments and results rather than migrating the mobile agent.

However, in many cases, agent mobility is a good choice. For example, the interaction of an agent with a large database can be better done by moving the agent to the provider host, and then locally interacting with the data. Note that database access by copy is unacceptable because it might use too much network bandwidth. Also, remotely querying the database may not be suitable for network latency reasons, specially when the number of queries to be performed is high.

In this sense, GRIM includes extra methods for accessing resources besides agent mobility. GRIM supports remote invocation for interacting with service-like resources, and replication of resources, for the case of files and data repositories. From the resource access point of view, this is like providing different ways of accessing resources or *extending* the RMF model. However, from the mobility point of view, we claim this is a *generalized* model of mobility since remote invocation implies control flow migration, and replication can be considered as a form of resource migration. The three forms of mobility considered by GRIM, as illustrated in Figure 5.4, are:

[2] Java Applets and ActiveX controls are examples of technologies based on this paradigm

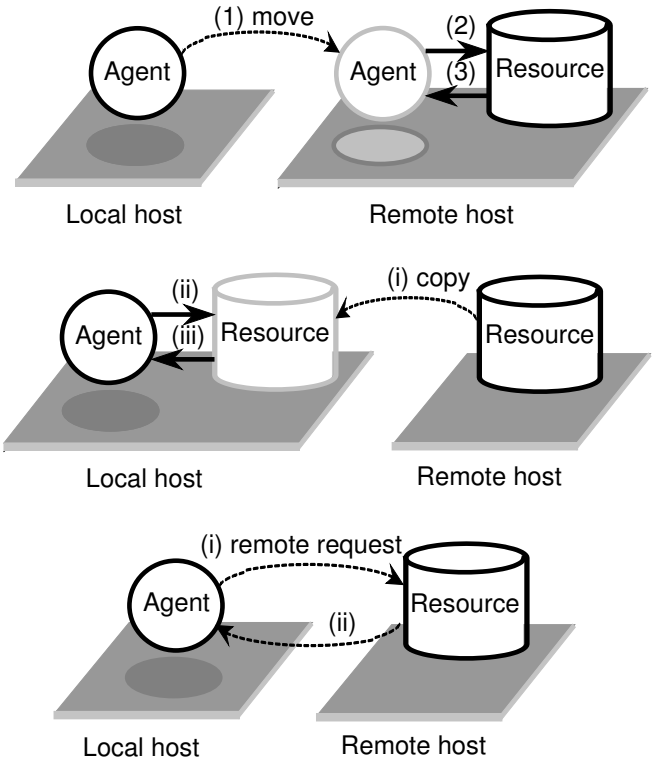


Figure 5.4: Forms of mobility in GRIM: agent, resource and control flow migration

- *Agent migration (move)*: As it name indicates, this method moves the agent to the host having the required resource. Once at the remote site, the agent is able to locally interact with the resource through its protocol. A typical scenario where agent migration can bring significant benefits is when accessing remote resources from mobile devices such as laptops and cell phones, which are usually subject to unreliable, low-bandwidth, high--latency wireless network connections (Vasiu and Mahmoud, 2004).

  At the middleware level, this can be supported by a *move* sentence implementing either a *strong* or a *weak* agent migration mechanism (Fuggetta et al., 1998) (see Table 5.1 for a brief comparison of these mechanisms). *Strong* refers to the ability of a mobile agent runtime system to allow migration of both the implementation code and the execution state of a mobile agent. In opposition, *weak* migration cannot transfer the execution state of agents. This approach has a negative impact in agent design, because an agent itself must be thought as a state machine to save/restore the execution state at the application level. The current materializations of GRIM (Zunino et al., 2005b; Mateos et al., 2007b), including JGRIM, are based exclusively on strong migration.

- *Resource migration (fetch)*: This form of mobility transfers the resource from the hosting location to the current agent's location, for example by copying it to a shared repository accessible to the agent. Resources such as data streams, structured files and code

	Strong migration	**Weak migration**
Migration of:	Implementation code and execution state	Implementation code only
Execution resumes from:	Next statement	A user-defined restart entry point
Enables:	Proactive and reactive migration	Proactive migration only
Benefits/drawbacks	(-) The *move* primitive is hard to implement (+) Better programming style (+) Agent code is easy to understand and debug	(+) The *move* primitive is easy to implement (-) "Unnatural" non-modular programming style. (-) Difficult to reason about and debug.

Table 5.1: Strong vs weak agent migration

libraries can be effectively replicated using this mechanism. The only requirement for a resource to be "fetchable" is that its protocol must include a *transfer(srcHost, dstHost)* operation, which implements the logic along with all technology-related issues for moving that resource between two hosts. This operation is invoked by the middleware whenever its runtime system decides to fetch the resource. At present, JGRIM implements this support for the case of file-like resources.

- *Control flow migration (invoke)*: The idea is to "migrate" the call requesting a resource by creating a new flow of execution on the remote host, blocking the requesting agent until an answer is received, and then resuming the normal execution flow. Technologies such as RPC and RMI are two examples of popular alternatives for supporting control flow migration at the middleware level.

  Control flow migration in JGRIM is supported through Web Service calls, as all resources are assumed to be wrapped by WSDL interfaces. Currently, this support is implemented by using the remoting and Web Services support provided by the Spring framework, and a subset of the functionality of Web Services Invocation Framework (WSIF), a Java API that allows to stubless and dynamically introspect and call Web Services based upon the examination of WSDL descriptions at runtime.

In GRIM, fetchable or *transferable* resources are those which can be moved or replicated from one host to another (e.g. files, data repositories and environment variables) whereas *non-transferable* ones are those which cannot (e.g. hardware components like printers and scanners). Furthermore, GRIM defines two kinds of transferable resources: free and fixed. Free resources can be freely migrated across different hosts. Moreover, fixed resources represent data and software components whose transfer is either not suitable (e.g. a very large database) or not permitted (e.g. a password-protected file). Figure 5.5 illustrates this taxonomy.

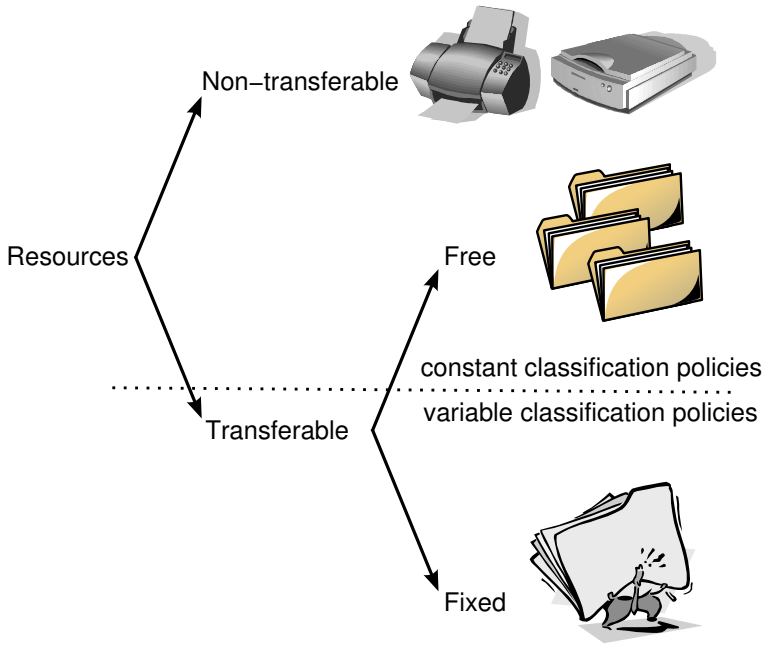


Figure 5.5: GRIM resource taxonomy

Transferability of free and non-transferable resources can be determined statically: the former are always transferable, whereas the latter are never transferable. On the other hand, note that transferability of fixed resources is determined at runtime, even when by nature this kind of resources might be able to be transferred. GRIM defines the concept of *classification policy*, which comprises the middleware-level logic in charge of dynamically associating a resource with a single leaf from the taxonomy tree of Figure 5.5. Resource publishers must provide a classification policy for every resource they add to a host. These policies tell the middleware whether a resource can be transferred or not.

Classification policies can be either *constant*, where the type for a resource is configured statically and do not vary along time (i.e. non-transferable or free transferable resources), or *variable*, where the type and therefore transferability of a resource is computed dynamically. For example, one might configure a simple sharing/replication scheme whereby transferability of a particular resource is subject to the identity of the host requesting the transfer.

5.2.1.2 Middleware-level policies

Deciding which form of mobility to employ in order to efficiently access distributed resources strongly depends on highly dynamic execution conditions such as network metrics, agent and resource size, available processing power and storage space, etc. In this sense, *middleware-level policies* (m-policies for short) allow GRIM hosts to statically configure decisions regarding which migration method to use upon traps on a protocol. For example, a host M_1 might set

an m-policy specifying that "all access to a resource R hosted at a host connected through an unreliable link should trigger agent mobility". This policy will affect any agent trying to access R when executing at M_1.

When more than one host offer the resource needed to handle a trap, it may be necessary (e.g. reliability reasons) to contact some or all of them to interact with their corresponding resource instance. Furthermore, the order for contacting these hosts may be important. For example, it may be convenient to contact the sites according to their speed, CPU load or availability. M-policies also let to define the destination for an agent when more than one destination is available for trap handling. In other words, m-policies provide criteria to select which resource to use from a set of candidates exposing the same GRIM protocol.

Besides simplifying mobile agent programming, a goal of GRIM is to provide a reference model for implementing *intelligent* middleware for mobile agents, automating mobility and access to resources in a clever way through platform-level policies. In the end, m-policies aim at configuring intelligent decisions for managing agent and resource mobility. However, "intelligent" here should be understood as making best-effort decisions, but not with the meaning that AI-related areas give to that term. In other words, GRIM does not explicitly prescribes any learning technique for managing mobility neither for agents nor resources.

5.2.1.3 Agent-level policies

GRIM allows agents to delegate decisions about from where and how to access resources, thus providing an elegant and convenient programming style for mobile applications. However, doing so implies that developers lose control of how these decisions are actually made. Under certain circumstances, this can be troublesome and lead to unacceptable application performance and bad use of resources.

Consider, for example, a situation where an agent located at a host M_1 needs to use a database D located at M_2. Let us assume that an m-policy for optimizing network traffic has been configured for M_1, and the agent has a significant size. As a result, every time the agent accesses D, a trap will be triggered, and then a control flow migration will be carried out. In the end, a remote query to the database is sent.

As far as it has been discussed, GRIM has problems with this type of situations because it does not take into account particularities about the application being executed. In our example, agent performance will be good or poor depending on whether the number of queries performed by the agent to the database is low or high, respectively. Generally speaking, this is caused since GRIM is not able to know whether the agent will carry out many or few interactions with the same resource within a given time frame.

To cope with this, GRIM introduces another type of policy called *agent-level policies* (a-policies for short), which allows developers to customize the way agents interact with resources. A-policies are defined separately from the agent's behavior, and they are attached to a protocol.

When binding after a trap, GRIM evaluates (if declared) the a-policy attached to the protocol causing the trap to decide the particular resource instance and the access method that should be used. A-policies have a greater priority than m-policies, in the sense that agent decisions overrides middleware ones for the same protocol.

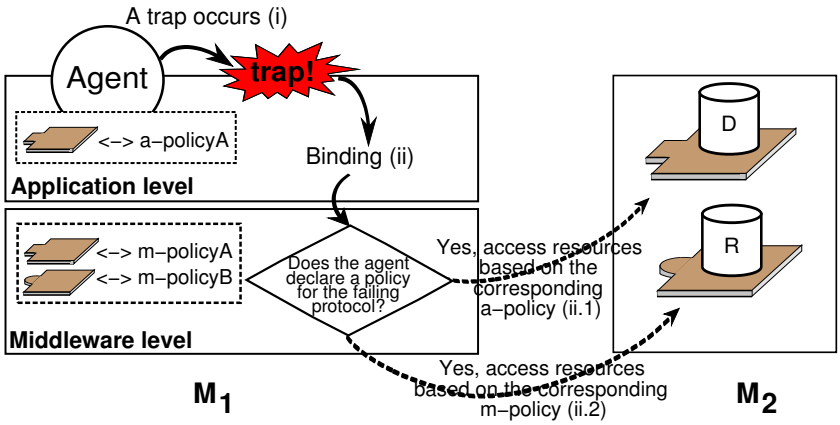


Figure 5.6: A-policies vs m-policies

Figure 5.6 illustrates the relationships between a-policies and m-policies. The middleware--level policy identified as *m-policyA* represents the policy for optimizing traffic explained above, whereas *m-policyB* represents access decisions associated to other resources, whose interface has been depicted in the figure with a semi-circle. In addition, the agent has declared an a-policy named *a-policyA* to act upon access to resources whose interface is the same as the database D, which has been represented as a square. In our previous example, a trap on the protocol associated to D will cause the middleware to access the database based on *a-policyA* rather than *m-policyA*.

To further clarify these ideas, we provide below a concrete implementation for *a-policyA* in JGRIM:

```
import jgrim.core.Policy;
import jgrim.core.Constants;
import jgrim.metrics.Profiler;

public class DataBasePolicy extends PolicyAdapter{

  public String accessWith(String methodA, String methodB){
    return Constants.MOVE;
  }

  public String accessFrom(String hostA, String hostB){
```

```
    Profiler pr = Profiler.getInstance();
    // The next two lines compute network latency from the
    // local host to hostA and hostB, respectively
    double latA = pr.profile("latency", pr.getLocalHost(), hostA);
    double latB = pr.profile("latency", pr.getLocalHost(), hostB);
    return (trA < trB) ? hostA : hostB;
  }
}
```

The `accessFrom` method defines the logic to select the desired resource instance from any pair of available candidates, whereas `accessWith` contains the code for choosing an access strategy given any pair of valid access methods for the database. The former returns the *move* method (ignoring other options), thus forcing the agent to migrate to M_2 upon the first query to the database D. Now let us suppose that a replica of database D is available at a third host M_3. Method `accessWith` ensures that our agent will access the remote host containing the database (M_2 or M_3) to which the lowest latency from the current agent's location is experienced.

As shown in the code, system metrics in JGRIM are obtained through the `Profiler` class, which wraps and enhances the services provided by the NWS (Wolski et al., 1999), a distributed system that monitors and forecasts the performance of a network and its nodes according to metrics such as CPU processing power and load, memory availability, network bandwidth and latency, among others. The next chapter provides details on the implementation of this support.

Policies can also redefine two more methods, not shown in the previous example, named `executeBefore` and `executeAfter`. As their name indicate, they are evaluated before and after calls to `accessFrom` and `accessWith` take place. These methods are particularly useful for implementing more complex policies. For example, one might implement a stateful policy that registers the performance of the access to certain services and uses this information to dynamically infer (e.g. by means of statistical methods or learning algorithms) the best way to interact with these services.

In short, a-policies let agents to define custom policies for adapting GRIM to fit specific application requirements. In other words, the "wild" version of GRIM might not be enough to make agent execution efficient under certain circumstances, therefore the mechanism must be "tamed". In fact, according to the Merriam-Webster Dictionary[3], "grim" means "savage", which in turn is defined as the condition of not being "domesticated or under human control". A-policies play the role of controlling GRIM despite the default behavior for managing mobility it had prior to agent execution.

[3]Merriam-Webster Online Dictionary: http://www.m-w.com

5.3 Conclusions

This chapter presented GRIM, a reference model for mobile computing in distributed environments. GRIM provides a conceptual framework that guides the materialization of mobile applications –and underlying middleware technologies– according to the idea that the code for handling mobility-related issues is, to a certain extent, orthogonal to the logic implementing agents. GRIM takes advantage of this fact by automating some decisions concerning the management of the non-functional behavior of mobile agents, therefore making mobile code programming simpler and easier.

GRIM has been already materialized by several middlewares for mobile application programming. Among them is JGRIM, a GRIM-based platform based on the Java language that also provides a materialization of the GRATIS approach to gridification. JGRIM further exploits the above orthogonality by conceiving the non-functional behavior of applications beyond mobility, and considering common Grid concerns such as service discovery, service invocation and parallelization. The next chapter describes the design and implementation of the JGRIM middleware.

Chapter 6

The JGRIM middleware

This chapter describes the design and implementation of the JGRIM middleware (Mateos et al., 2008), which has been introduced in previous chapters. Particularly, the description will put emphasis on explaining the implementation of the mechanisms that are employed by JGRIM to enable the transparent injection of Grid services to Java applications. The chapter includes some diagrams in Unified Modeling Language (UML) notation to better understand the various components of the middleware and the way they interact with Grid services.

The chapter is structured as follows. The next section briefly introduces the JGRIM platform. Then, Section 6.2 presents the general architecture of the JGRIM runtime environment and discusses the most relevant aspects related to its design and implementation details. Finally, Section 6.3 concludes the chapter.

6.1 The JGRIM Platform

The execution context for gridified applications provided by a host of a JGRIM-enabled network is based on HTTP Java servlets (Hunter and Crawford, 1998). Basically, an HTTP servlet is a stationary object accessible through the HTTP protocol that implements two methods: `doGet` y `doPost`. These methods are invoked by the Web or application server in which the servlet is running upon a client request. Typically, parameters passed in to `doGet` and `doPost` indicate the particular service that must be served by the servlet. In JGRIM, these services include MGS deployment, resource monitoring, protocol exchanging, and so on. The only difference between `doGet` and `doPost` is the way request data is sent from the client to the servlet. However, `doPost` is able to handle large values for parameters (e.g. a moving agent from another host), whereas `doGet` cannot. The reason for which the Java servlet technology was chosen is that it is currently supported by almost all Web servers.

A JGRIM-enabled host can belong to one or more logical networks. A host within a specific logical network is identified through a universal locator of the form `http://address:port/servlet`,

where *address* is the machine name or IP address of the host, *servlet* is the name of the corresponding servlet, and *port* is an integer value is the identifier of the logical network. Essentially, logical networks serve as a mean to establish many service networks (e.g. with different computing capabilities) among hosts belonging to the same or even different administrative domains.

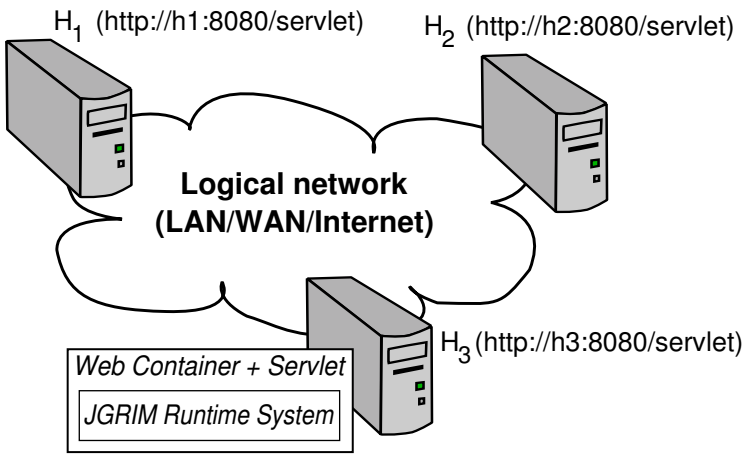


Figure 6.1: Overview of the JGRIM platform

Figure 6.1 summarizes the concepts mentioned above. As observed from the figure, a JGRIM logical network (from now on simply a network) is composed of a number of physical nodes, which are globally identified through an URI. Furthermore, each node is equipped with a software support that provides an execution environment for (gridified) JGRIM applications. The next section describes this support in detail.

6.2 The JGRIM Runtime System

Figure 6.2 depicts the architecture of the JGRIM runtime system. Basically, the components of this architecture are organized in three layers: the *Application* layer, which represents gridified applications or MGSs, the *Core API* layer, which provides access to concrete Grid services by means of built-in Grid metaservice components, and the *Injection* layer, which is in charge of transparently and seamlessly binding applications and metaservice components.

The Injection layer is implemented based on the facilities provided by Spring (Walls and Breidenbach, 2005; Johnson, 2005), a Java-based, lightweight framework for the development of enterprise grade distributed applications. Spring is a highly modular framework composed of seven modules that implement functionality such as remote access to applications and services

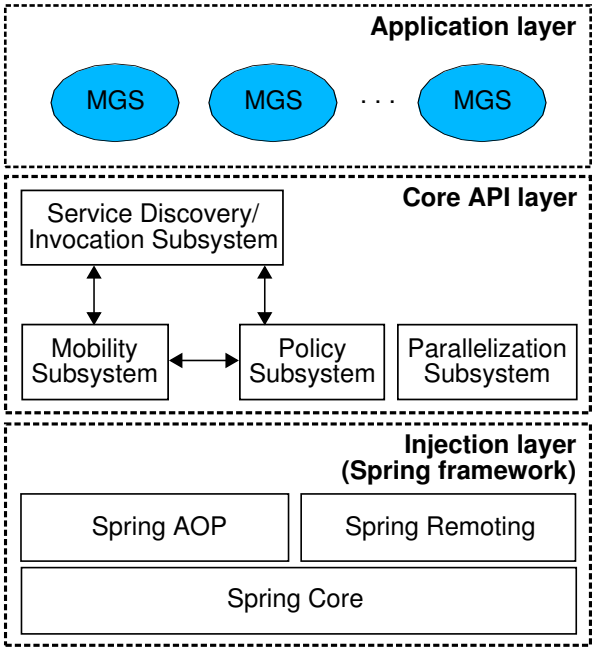


Figure 6.2: Architecture of the JGRIM runtime system

(e.g. RMI or SOAP services), communication and messaging, declarative transaction management, data persistence, and so on. Because of its good modularity, Spring allows developers to selectively use only those modules that are required by their applications. In fact, as illustrated in the figure, JGRIM only make use of the Spring Core, Spring AOP (aspecting support) and Spring Remoting (remote service invocation) modules. Nevertheless, programmers who are familiarized enough with Spring can further exploit it by using the rest of the functionality. In this way, applications can simultaneously take advantage of both JGRIM metaservices and Spring components.

The core of Spring's design is the Spring Core module, which provides a consistent mean of managing Java objects and their associated dependencies. In other words, this module implements a Dependency Injection container for Java applications. Objects created by the container are entities exposing well-defined interfaces called *JavaBeans* (or simply *Beans*). A Spring application is essentially a collection of decoupled bean components that interact between each other according to the DI pattern. Certainly, there are quite a few Java frameworks for implementing DI-based applications (e.g. Apache's HiveMind[1], Google's Guice[2], PicoContainer[3], Seasar[4], etc.), but Spring has become widely popular in the Java community since it offers a lot of freedom to developers yet provides well-documented solutions for common practices in the industry. In addition, unlike other frameworks, Spring is not just a DI container, but a complete enterprise development platform.

[1]HiveMind: http://hivemind.apache.org
[2]Guice: http://code.google.com/p/google-guice
[3]PicoContainer: http://www.picocontainer.org
[4]Seasar: http://www.seasar.org/en/index.html

On top of the Spring facilities, the *Core API* layer materializes metaservice components (i.e. m-components) through specialized beans that either wrap existing concrete Grid services or implement new ones. These beans play the role of the injected components of gridified applications. Roughly speaking, metaservice beans are grouped in four different subsystems, each representing a different kind of Grid metaservice:

- *Service Discovery/Invocation subsystem*: Beans at this subsystem are in charge of dealing with Web Service discovery and invocation across a Grid, that is, providing concrete bindings between ordinary applications and external service components. Currently, service discovery is supported by inspection of UDDI registries, while service invocation is performed by extending and enhancing the components of the Spring Remoting module. The Service Discovery and Invocation subsystem is described in Section 6.2.1.

- *Policy subsystem*: Both user and middleware-level policies are represented by special beans that provide an extensible support for specifying decisions related to application tuning. At runtime, service discovery and invocation beans may talk to policy beans to customize the way Grid services are accessed. Similarly, customized pre and post actions attached to dependencies on internal application components are also represented by means of policy beans. The class design and details on the implementation of this subsystem are discussed in Section 6.2.2.

- *Mobility subsystem*: Basically, mobility-related beans offer strong migration services to gridified applications. Although JGRIM migration primitives may be used explicitly by developers in the application code, mobility services are mainly intended to be accessed by means of user policies, which are configured separately from the application logic. Section 6.2.3 describes the implementation of this subsystem.

- *Parallelization subsystem*: Metaservice beans of the parallelization subsystem materialize the necessary support to associate concrete method infrastructure-level execution and spawning services to self-dependencies. Parallelization in JGRIM is currently based on the services provided by the Ibis platform (Section 3.1.9). Section 6.2.4 explains the implementation of the parallelization subsystem. As we will see later, the JGRIM parallelization components can be easily extended to leverage other execution services such as those provided by ProActive or Globus.

6.2.1 The Service Discovery/Invocation Subsystem

The Spring Remoting module provides a programming abstraction for working with various RPC-based service technologies available on the Java platform, both for client connectivity and exporting objects as services on remote servers. This module isolates applications from the intricate configuration and coding details involved in using these technologies. Spring achieves this separation by proxying remote services with special beans that rely on the component

injection and aspecting features of the Core and AOP modules, respectively. Basically, these beans are responsible for decoupling client applications from services.

Among those beans is the `JaxRpcPortClientInterceptor`, which provides transparent access to Web Service operations by means of JAX-RPC, a protocol to implement and call Web Services described in WSDL. An application can define dependencies to particular Web Services by supplying the corresponding interfaces for these dependencies and the contact information of the services. For example, the following code shows the Spring configuration of an application declaring a dependency to an accounting Web Service:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="client" class="example.AccountClientImpl">
    <property name="service">
      <ref bean="accountWebService"/>
    </property>
  </bean>
  <bean id="accountWebService"
    class="spring.JaxRpcPortProxyFactoryBean">
    <property name="portInterface">
      <value>example.AccountService</value>
    </property>
    <property name="wsdlDocumentUrl">
      <value>http://localhost:8080/account?WSDL</value>
    </property>
    <property name="namespaceUri">
      <value>http://localhost:8080/account</value>
    </property>
    <property name="serviceName">
      <value>AccountService</value>
    </property>
    <property name="portName">
      <value>AccountPort</value>
    </property>
  </bean>
</beans>
```

where *wsdlDocumentUrl*, *namespaceUri*, *serviceName* and *portName* are the contact information of the service, and *portInterface* is the contract to interact with the service. `JaxRpcPortProxyFactoryBean` is the Spring factory class that creates an instance of a proxy for the service. Then, this instance is injected into the `AccountClientImpl` in order to transparently translate

any method call performed on the dependency interface (`AccountService`) to the corresponding Web Service operation.

JGRIM extends the above support to get rid of the configuration details for contacting individual services, and otherwise to extract this information from service registries. The `WSDLMatcher-PortClientInterceptor` (Figure 6.3) implements a service proxy that searches for Web Services that match the interface of its associated dependency. This process involves two more classes, namely `GenericWSDLFinder`, which is a proxy to the service registry (currently UDDI) where the service descriptions are stored, and `WSDLMatcher`, which implements the logic to determine, from a WSDL description, whether a specific service matches the operations of a Java interface. Programmers can easily refine the matching process by specifying custom mappings between application datatypes and WSDL types. In GRATIS terminology, the `WSDLMatcherPortClientInterceptor` class is a materialization for the m-component of the approach in charge of injecting functional Grid services to applications.

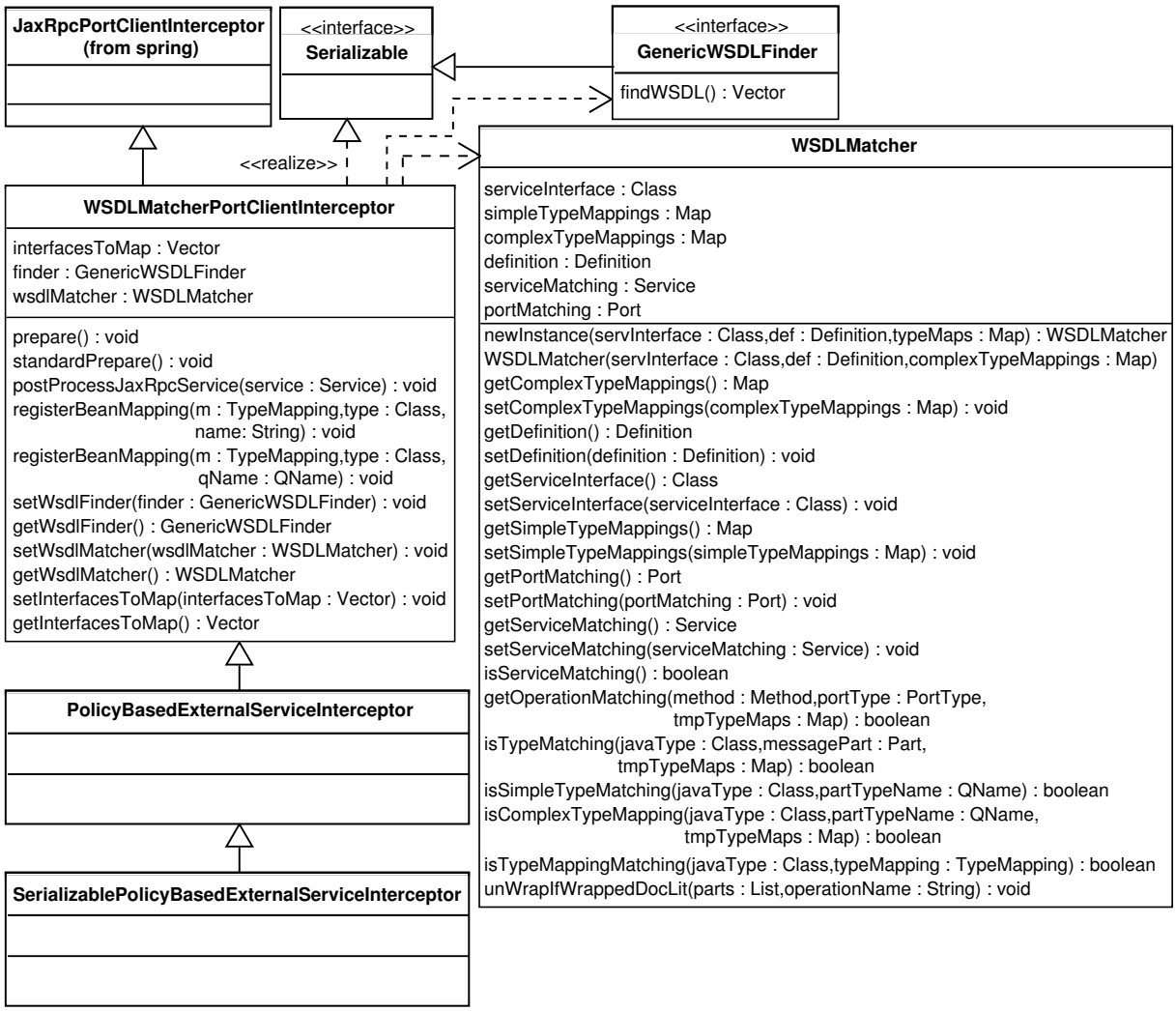


Figure 6.3: Class design of the Service Discovery/Invocation subsystem

Another important class in the diagram is `PolicyBasedExternalServiceInterceptor`, which implements a service discovery bean that contact Web Services based on policies. As such, the bean uses the methods of the policy bean configured for accessing the service. After querying a UDDI registry, a discovery bean passes on the list of available candidates to its associated policy bean to find out which service instance should be used, and how it should be contacted. As a consequence, a discovery bean may, for example, trigger the migration of the application to the host where the service instance is hosted, or ask the external service –in case it is an MGS– to migrate to the local host. In this sense, the `SerializablePolicyBasedExternalServiceInterceptor` class implements the support for using policy-based service discovery and invocation beans in conjunction with mobility beans.

Every JGRIM service proxy have a reference to a *finder* bean that implements the `GenericWSDLFinder` interface. Generally speaking, finder beans are responsible for retrieving Web Service metadata from service registries based on pre-configured search criteria. As mentioned above, JGRIM provides built-in support for querying UDDI service registries. Figure 6.4 illustrates the class diagram of the UDDI discovery support of JGRIM. For space reasons, some classes have been suppressed its methods. Each UDDI finder represents a specific search criteria to inspect the relational model that UDDI maintains to describe Web Services. To simplify the usage of discovery metaservices, JGRIM injects by default business finder beans, which search for Web Services based on a case-sensitive match on the owner of services. However, application developers have the freedom to employ other inspection mechanisms as needed by attaching other types of finder bean to service proxies.

The `UDDIProxy` class is a proxy for a UDDI registry that is part of the UDDI4J library[5]. Basically, instances of this class provides clients with the ability to utilize any of the API methods described in the UDDI API specification. Furthermore, `UDDIProxyWrapper` is a class provided by the JGRIM API that implements a lazy initialization mechanism for the instances of the `UDDIProxy` class to minimize the objects that are transferred during application and therefore proxy migration. The parameters needed to establish a UDDI session –address of the UDDI registry, authentication information and communication transport protocol– are supplied by the JGRIM runtime. Again, developers can override these parameters by modifying the configuration of the UDDI proxy beans in the XML file of a gridified application.

The next subsection describes the Policy subsystem.

6.2.2 The Policy Subsystem

The class hierarchy materializing the Policy subsystem of JGRIM is shown in Figure 6.5. The root of the hierarchy is the `Policy` class, which represents a generic policy component that

[5]The UDDI4J API: http://uddi4j.sourceforge.net

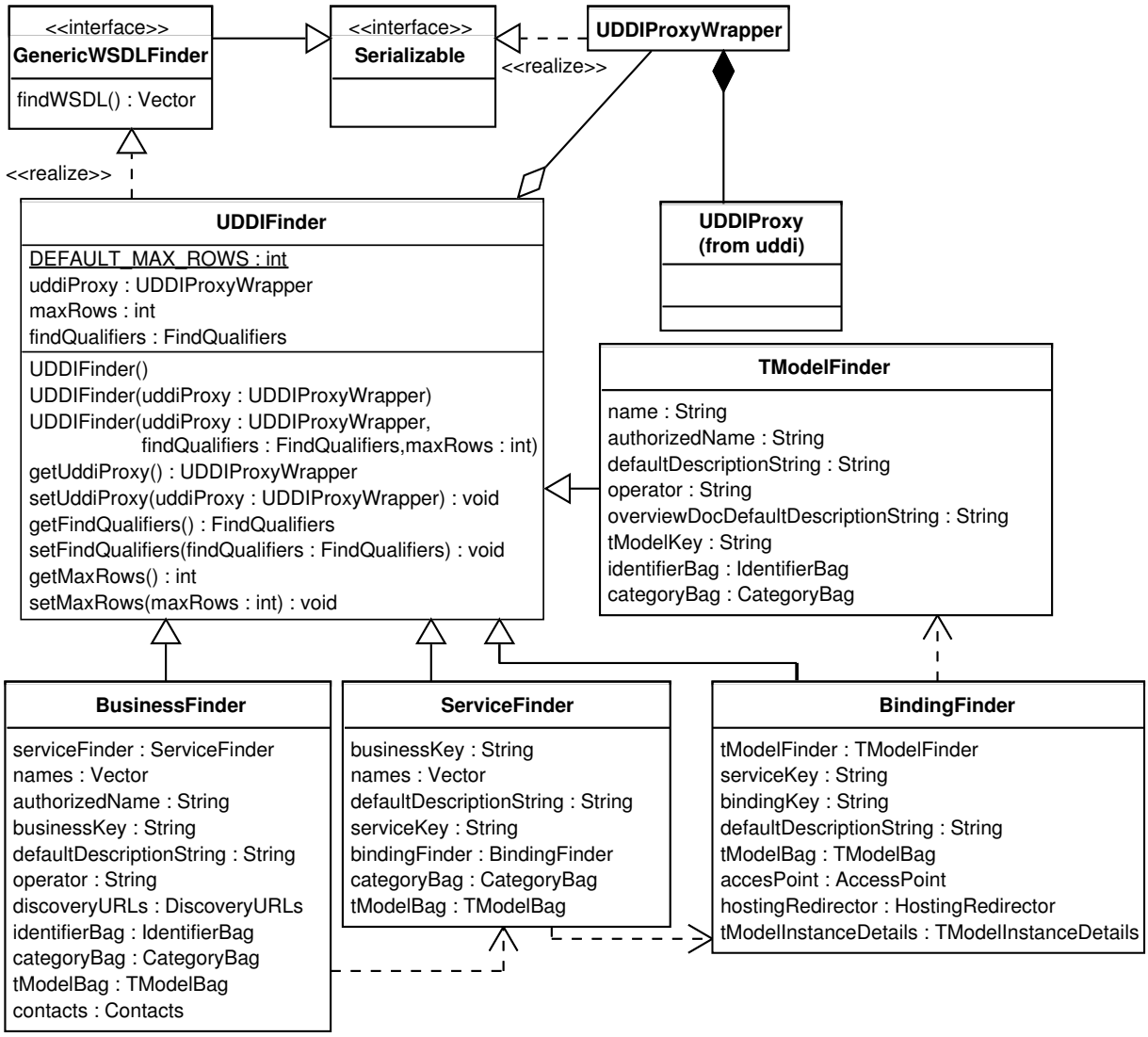


Figure 6.4: Classes materializing the inspection of UDDI registries

can act before and after a method of a particular dependency is invoked. In this sense, the methods `executeBefore` and `executeAfter` must be implemented by its subclasses in order to specify the concrete set of actions that will be executed upon an invocation of any method of the dependency to which the policy is associated. In addition, policies can be temporarily activated/deactivated by implementing the `isActivated` abstract method. Lastly, all policies have a reference to its owner agent, and another reference to a Spring object (instance of the `BeanFactory` class) that keeps track of all beans owned by the agent. In this way, policies can access the methods of any application component within an MGS, which leads to flexibility and reusability of components.

The `PolicyAdapter` class simply provides a default (empty) implementation for the `executeBefore`, `executeAfter` and `isActivated` methods. It also includes some extra methods to

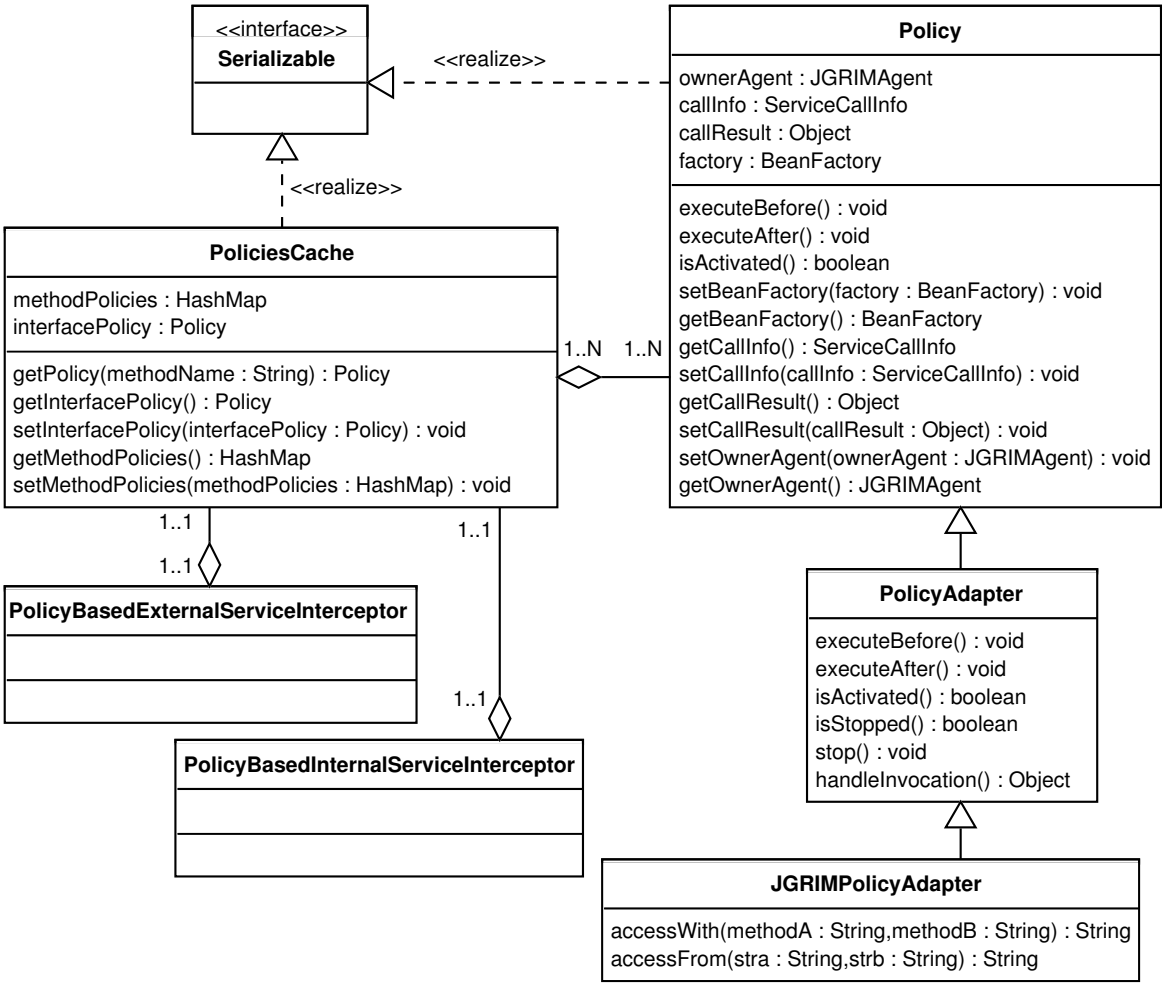


Figure 6.5: Class design of the Policy subsystem

permanently cease the activity of the policy (`stop`), and to manually handle the invocation of a dependency method (`handleInvocation`). For example, one might implement a policy to cache the results of the invocation to methods of either external or internal components, thus improving performance. In this case, `handleInvocation` should be implemented in such a way that some invocations (e.g. those with certain values for the arguments) receive a return value extracted from a caching structure maintained by the policy. By default, not overriding this method means that JGRIM will carry out the invocation directly on the target component. Instances of `PolicyAdapter` are used by the `PolicyBasedInternalServiceInterceptor` class, whose instances represent dependencies between internal application components.

Other important classes of the Policy subsystem are `JGRIMPolicyAdapter` and `PoliciesCache`. Basically, the former represent a full-fledged JGRIM policy that can be attached to an external service component to decide how to interact with concrete functional services, and perform customized actions before/after invoking operations of these services. Instances of `JGRIMPolicyAdapter` are used by the `PolicyBasedExternalServiceInterceptor` class, whose instances represent dependencies from an application component to an external component.

Furthermore, the `PoliciesCache` class basically implements a policy registry, which is composed of one or more policies affecting the invocation of methods on a particular dependency interface. The *interfacePolicy* attribute represent (if defined) the policy controlling the entire dependency interface, whereas *methodPolicies* is a map containing a number of policies that act upon invocation on specific methods of this interface. To clarify this aspect, below is a skeleton of the XML configuration generated for a gridified application employing both interface and method policies on a dependency whose interface is `MyInterface`:

```
...
<bean id="1"
   class="core.discovery.PolicyBasedExternalServiceInterceptor">
  <property name="expectedInterface">
    <value>MyInterface</value>
  </property>
  <property name="policiesCache">
    <ref bean="cache_1"/>
  </property>
</bean>
<bean id="cache_1" class="core.policy.PoliciesCache">
  <property name="interfacePolicy">
    <!-- Reference to a policy bean -->
    <ref bean="2"/>
  </property>
  <property name="methodPolicies">
    <map>
      <entry>
        <key>MyInterface.someOp</key>
        <!-- Reference to another policy bean -->
        <ref bean="3"/>
      </entry>
    </map>
  </property>
</bean>
...
```

On one hand, the policy bean identified as "3" is in charge of controlling the invocations performed on the operation `someOp` of `MyInterface`. For simplicity, method overriding has been omitted. On the other hand, invocations to the other methods of `MyInterface` are intercepted by the policy bean whose identifier is "2".

6.2.2.1 Profiling

The basic elements upon which policies are usually built are system metrics. JGRIM offers the programmer a number of API methods whose purpose is to return accurate values for metrics such as CPU load, free memory, network performance, among others. The API is accessed through the `Profiler` class, which is a singleton component providing a well-defined profiling interface for which many different concrete implementations could be supplied.

To date, profiling in JGRIM is based on NWS (Wolski et al., 1999), a distributed monitoring service that periodically measures and predicts the performance of both network and computational resources. The service operates and controls a distributed set of performance sensors (e.g. network monitors, CPU monitors, memory monitors, etc.) from which it gathers readings of the instantaneous conditions. NWS then uses numerical models (mean-based, median-based and autoregressive) to generate forecasts about the conditions for a given time frame.

Because NWS is highly dependent on the operating system platform, and it has proved to be difficult to deploy on Grid settings, another profiling component based on JMX[6] is being implemented. JMX is a Java-based API for building distributed solutions for monitoring devices, applications and networks. Diffusion of metric information will be performed by means of GMAC (Gotthelf et al., 2005), a protocol specially designed to provide Internet-wide multicast services to mobile agent platforms.

Programmers can use the JGRIM profiling API to code complex heuristics to improve application performance. Specifically, the services exposed by the API include:

- $load(S)$, which returns a forecast of the average CPU utilization (percentage) at the site S. It is a useful measure under certain circumstances. For example, if an expensive operation of an internal application component needs to be executed, and the local CPU load is twice the CPU load of a remote site R, the MGS may be forced to move to R, thus providing a simple tactic for load balancing across sites. A service similar to *load* is provided to obtain statistics about memory usage.

- $agentsAt(S)$, which computes the number of executing agents at S. Note that this metric is closely related to CPU load, especially when agents are doing CPU-intensive work, that is, they are neither blocked nor waiting for some notification or success. In addition, *agentsAt* gives an approximate idea of both the CPU and memory utilization.

- $agentSize(A)$, which returns the estimated size (in bytes) of the allocated memory space for an executing agent (i.e., the allocated RAM for its object graph). It is a useful metric to determine whether it is appropriate to migrate an agent, and it can be used together with metrics that estimate the network conditions.

[6] Java Management Extensions (JMX) Technology: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement

- *latency*(S_1, S_2), which represents the estimated latency (in seconds) for transmitting data from a site S_1 to another site S_2. Having measures about network latency is crucial to decide, for example, which external service to contact from a set of available candidates. Similarly, the profiling API also provides a service to estimate the network transfer rate (measured in KB per second), and the communication reliability, that is, the percentage of the information lost during transference per unit time.

The subsytem materializing MGS mobility is presented in the next subsection.

6.2.3 The Mobility Subsystem

During gridification, the original code of the entry point class of a Java application is automatically modified in order to inherit the behavior of the `JGRIMAgent` class, which gives birth to an MGS. Basically, this class provides basic primitives for handling agent mobility and managing agent state information (i.e. application-specific knowledge that is private to the agent). As mentioned above, these primitives are particularly intended to be invoked from within policies, so as to keep the application logic away from the JGRIM API. The class design of the Mobility subsystem is shown in Figure 6.6.

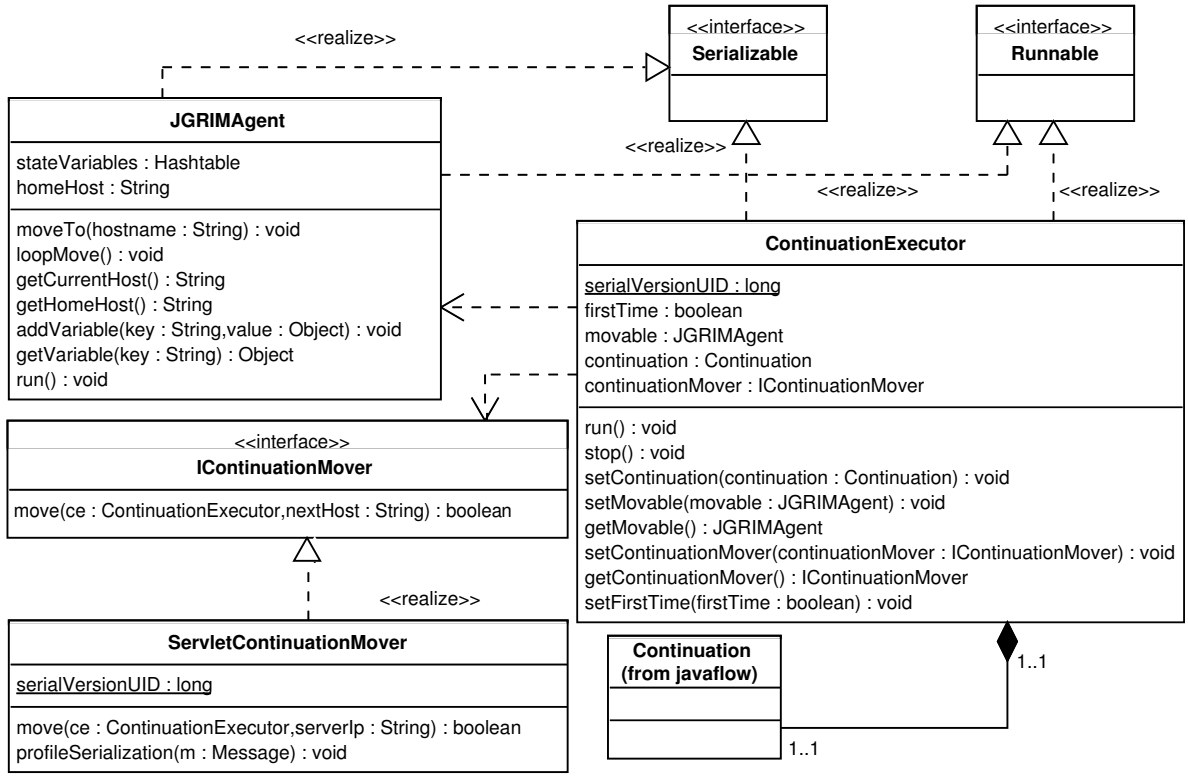


Figure 6.6: Class design of the Mobility subsystem

The `JGRIMAgent` class mostly provides methods that implement mobility-related functionality. Specifically, methods `getCurrentHost` and `getHomeHost` return the address of the host where an agent is currently executing, and the site where it was originally deployed, respectively. Every policy declared by the original application is injected a reference to the corresponding instance of `JGRIMAgent`, thus both mobility and agent state primitives can be directly accessed by policies. Lastly, the `run` method contains the logic to transform an external service request –coming, for instance, from another agent or a client application– to the corresponding method call.

The `moveTo` method moves an agent to a specific host. When this primitive is invoked, the execution of the agent is frozen and stored in a `Continuation` class. A continuation is an object that, for a given point in the execution of the agent, contains a snapshot of the stack trace, local variables and program counter. This information is used to restore the execution of the suspended agent once it has been successfully migrated to the requested remote site. Continuation support was implemented through JavaFlow (Apache Software Foundation, 2006), an open source library that allows to capture the execution state of threads within a Java application. Basically, JavaFlow operates by instrumenting the bytecode of certain application classes in order to make the execution state accessible to the application. The reader should recall that, as for the current version, Java does not allow applications to access the full execution state of a running thread.

Each instance of `JGRIMAgent` is assigned an instance of the `ContinuatorExecutor` class. This class implements the execution services that are necessary to transfer continuation objects to remote hosts and resume their execution. Instances of the `ContinuatorExecutor` class are in turn assigned an instance implementing the `IContinuationMover` interface. Classes implementing this interface represent specific mechanisms for transferring continuation objects from a host to another (e.g. sockets, RPC, RMI, etc.). Because the JGRIM execution environment for MGSs at any host is currently based on servlets, the `ServletContinuationMover` class is responsible for marshalling agents into a byte array format, open TCP connections to remote servlets, and perform the reliable transfer of this data. Figure 6.7 illustrates the interacting classes involved in the process of initiating, suspending and resuming the execution of a JGRIM agent.

Apart from transparently moving its execution state, JGRIM also transfers the code of an agent in case this code is not present in the destination host. Specifically, when a host receives an incoming continuation object, it requests back to the origin host the missing bytecode that are necessary to fully restore the execution of the continuation. The server side processes continuation objects coming from the underlying network data stream by using a special class loader (implemented by the `NetworkClassLoader` class) that requests back missing classes as the continuation object is unmarshalled. Transferred .class files are stored on disk thus they can be sent to other hosts as well. In consequence, the deployment of the MGS implementation across hosts is done in a on demand basis and in a transparent way, thus saving bandwidth consumption and not involving developer intervention.

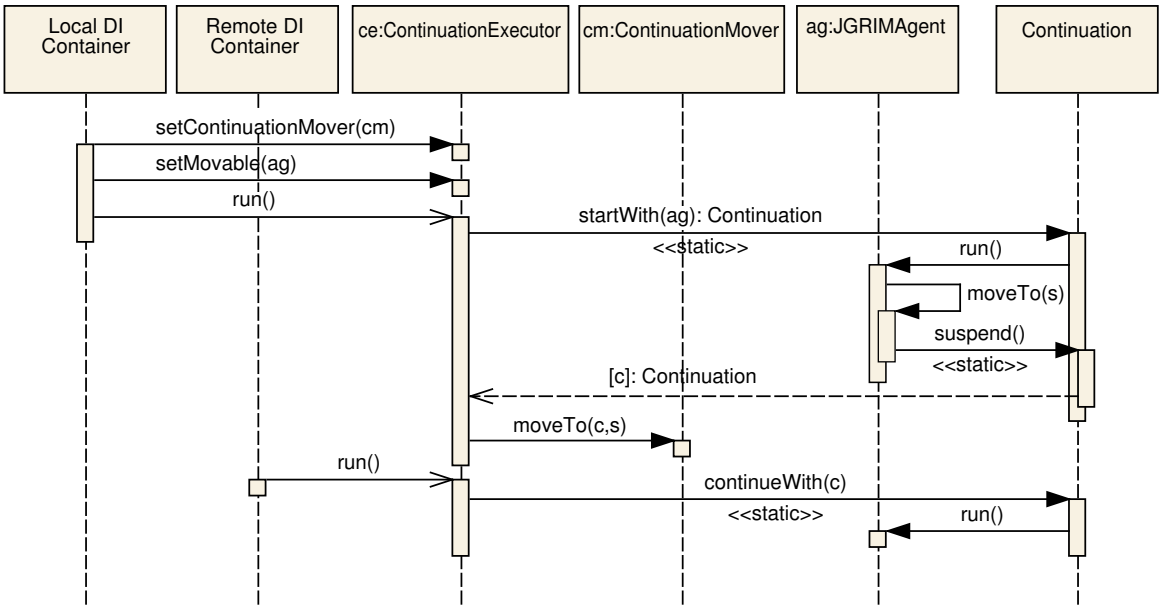


Figure 6.7: Initiating, suspending and resuming agent execution: sequence diagram

A problem that arisen when combining mobility with Spring beans was concerned with serialization. Java allows objects to be transferred through a network provided the classes associated with these objects implement the `Serializable` interface. Since both JGRIM and Spring make extensive use of classes which were not thought to be serializable, such as those contained in the Java reflection API or the Spring AOP module, marshalling agent execution state through the standard Java serialization support was not possible. To overcome this limitation, a generic serialization support for converting any Java class into a serializable one was implemented. Roughly, this support works by dynamically modifying the structure of an object (i.e. the associated bytecode) in order to force it to be serializable. In addition, this transformation is recursively performed on the instance and class attributes of the object, thus guaranteeing that the whole object graph representing an executing agent is serializable. It is worth mentioning that the implementation of this mechanism is completely independent of the JGRIM API, and only adds a minimum performance overhead with respect to the standard Java serialization mechanism. Therefore, the support can benefit non-JGRIM applications and other Java frameworks as well.

The next subsection discusses the implementation of the Parallelization subsystem.

6.2.4 The Parallelization Subsystem

The parallelization metaservices of JGRIM are based upon the concept of *interceptor*. An interceptor is a reified object that inserts behavior between a caller and a callee or, in others words, a method invoker and the object implementing the method. Interceptors are the key of AOP programming. Spring's AOP module has a rich, extensible support for AOP, and provides built-in interceptors to transparently apply cross-cutting concerns such as logging, security and profiling to an existing Java application without source code modification.

Spring materializes interceptors as beans that are commonly attached to application dependencies. Specifically, if an application component or bean A has a dependency to a another bean B, an interceptor can be configured to act upon method calls from A to B, and do something in between. However, this is transparent to the application, since the code implementing A and B remains untouched. For instance, Spring interceptors were used to partially implement the support for service proxies and policies discussed in previous sections, where A represents a client application bean, B is either a external or an internal service component, and the interceptor is the proxy itself (which can depend in turn on policy beans). In the case of self-dependencies, A is equal to B, that is, the caller and callee are exactly the same component.

To configure a self-dependency, a programmer must first define the interface of the dependency, which will be composed of the methods whose execution is to be handled by a JGRIM execution interceptor. Then, just like any other dependency, access to any of these methods within the application code should be done by using the self-dependency instead of calling the method directly. In case of invocations to non-void methods, a coding convention to store an invocation result similar to that of Ibis must be used. For example, given the code:

```
public class MyApplication{
  public void methodA(){
    ...
    if (methodB()){
      ...
    }
    ...
  }
  public boolean methodB(){...}
}
```

the user must perform two simple modifications to the original code, so as to place the result of the invocation to `methodB` in a local variable, and accessing a fictitious *self* dependency by means of the associated getter. Note that developers can avoid accessing the self-dependency in certain parts of the code by simply invoking `methodB` on *this* instead of doing so on the object returned by `getself`. Furthermore, the dependency interface must be clearly defined, thus resulting in the following code:

```
public interface SelfInterface{
  public boolean methodB();
}

public class MyApplication{
  public void methodA(){
    ...
    boolean result = getself().methodB();
```

```
    if (result){
      ...
    }
    ...
  }
  public boolean methodB(){...}
}
```

JGRIM then takes this code and automatically makes `MyApplication` to inherit from the `JGRIMAgent` class, and adds an instance variable of type `SelfInterface` and the necessary get/set accessors to `MyApplication`. Execution of self-dependency methods are handled concurrently with the execution of the caller method. In this sense, JGRIM also modifies the body of the methods that invoke operations on the self-dependency (in this case `methodA`) in order to insert one or more calls to a synchronization primitive (barrier) to block the execution until the results of concurrent computations are available (in our example, when the *result* variable is instantiated). All of these transformations are performed by using java2xml[7], a Java API for generating an XML representation of Java source code. Representing the application code as an XML document permits to utilize XPath[8], an XML query and manipulation language that allows to easily address and transform portions of XML documents.

At runtime, the actual execution of the operation is transparently delegated to an execution interceptor bean, whose definition is appended to the file that configures the various meta-service beans of an application. Basically, an execution bean knows how to intercept and forward a method execution request to an existing resource management system (e.g. Globus, Condor, Ibis). As a consequence, each execution bean implements the specific communication protocol to interact with a particular resource management system. Returning to our example, the generated configuration for beans would be:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="service" class="example.MyApplication">
    <property name="self">
      <ref local="self"/>
    </property>
  </bean>
  <bean id="self" class="spring.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>example.SelfInterface</value>
    </property>
```

[7]The java2xml Project: https://java2xml.dev.java.net
[8]The XPath Language Specification: http://www.w3.org/TR/xpath

```
    <property name="interceptorNames">
      <list>
        <value>interceptor</value>
      </list>
    </property>
  </bean>
  <bean class="jgrim.IbisInterceptor" id="interceptor">
    <property name="ownerAgent">
      <ref bean="service"/>
    </property>
  </bean>
</beans>
```

As shown in the above XML code, the entry point bean of the application (identified as *service*) declares a dependency to a *self* bean, which is a factory class that returns an AOP proxy to *interceptor*. AOP proxies are able to make any object to dynamically implement one or more Java interfaces. Invocations to any of the methods of the `self` object (i.e. those included in the `SelfInterface` interface) are intercepted by the AOP proxy and delegated to *interceptor*. Roughly, interceptors implement one operation:

```
public Object invoke(MethodInvocation method)
```

to provide custom code and, optionally, proceed with the execution of the actual method code (i.e. the one implemented by the *service* bean) based on the information received in the *method* argument. Currently, JGRIM supports only Ibis-based interceptors, which are implemented through the `IbisInterceptor` class. However, the separation achieved by JGRIM between a method invocation and the environment where the execution of the method is actually carried out serve as a hook on which other types of interceptors may be "hung". For instance, JGRIM provides a default interceptor that just redirects all method invocations back to the main application bean. This is useful to easily and straightforwardly remove the Grid parallelization services injected into the gridified application.

6.2.4.1 The Ibis Execution Bean

An Ibis execution bean is responsible for handling the execution of those operations within the application that are implemented under the divide and conquer paradigm. The main goal of this support is to take advantage of the sophisticated parallelization and load balancing services of the Ibis platform, but without the burden of modifying the application to use the underlying Ibis API (i.e. performing special subclassing, creating marker interfaces and using explicit synchronization primitives). The only requirement imposed to the programmer, which is a consequence of using Ibis, is that the results of recursive calls must be placed on local variables. However, this is a simple coding convention that does not involve Grid API usage at all.

To execute an application on the Ibis platform, the implementation of the application must obey certain structure conventions, namely subclassing a specific API class, and implementing a user-supplied interface that indicates the methods whose execution must be spawned. In consequence, the process of injecting Ibis services into an ordinary Java class involves the creation of another class (from now on the *peer*) whose code is automatically derived from the original class but transformed in such a way it adheres to the Ibis application structure. In addition, the implementation code associated to the peer should include calls to the Ibis `sync` primitive at proper places. Overall, the peer represents the Ibis-aware version of the gridified application code, which is indirectly accessed through a JGRIM execution interceptor.

Recall the application that was presented as an example at the beginning of this section. The application is implemented through a `MyApplication` component declaring a self-dependency on a single `methodB` operation. Let us additionally suppose that `methodB` implements a compute--intensive divide and conquer algorithm, thus it is a suitable code to be executed by Ibis. The peer class created by JGRIM is as follows:

```
public interface SelfInterface_Peer extends ibis.satin.Spawnable{
  public boolean methodB();
}

public class MyApplication_Peer extends ibis.satin.SatinObject{
  private byte[] serializedAgent = null;
  private transient Object targetAgent = null;

  public void methodA(){...}
  public boolean methodB(){
    ...
    boolean aBranch = methodB();
    boolean anotherBranch = methodB();
    ...
    // Ibis sync primitive is automatically inserted
    super.sync();
    return (aBranch || anotherBranch);
  }
  public void setSerializedAgent(byte[] serializedAgent){
    this.serializedAgent = serializedAgent;
  }
  public Object getTargetAgent() throws Throwable{
    if (this.targetAgent == null){
      // Deserializes the contents of "serializedAgent"
      // and assigns the result to "targetAgent"
    }
```

```
    return targetAgent;
  }
}
```

Since it is composed of spawnable calls to itself, the code implementing `methodB` is modified to include appropriate calls to the `sync` Ibis barrier primitive to ensure that the results returned by recursive calls are always available before they are used. In some cases, automatic insertion of synchronization primitives may lead to performance degradation, but it drastically simplifies application programming, and also isolates developers from the parallel programming API. Therefore, gridification is transparent and easy.

The *serializedAgent* variable holds the serialized version of the executing JGRIM agent accessing the self-dependency, that is, the `MyApplication` class. It is injected into the peer in order to transparently allow its code to keep using the dependencies and the state of the associated agent. For efficiency reasons, the agent is loaded in its object form on an on demand basis. In addition, upon being migrated by the Ibis runtime, only the data pointed by the *serializedAgent* variable is transferred because *targetAgent* is `transient`. Furthermore, getters corresponding to other application dependencies (e.g. external services) are rewritten to access –via the Java reflection API– their associated injected runtime object. For example, the extra code that is generated in the peer class if the agent now declares a dependency named *somedependency* is:

```
public SomeDependencyInterface getsomedependency(){
  return getDependency("somedependency");
}
protected Object getDependency(String name) throws Throwable{
  Object agent = getTargetAgent();
  Method m = agent.getClass().getMethod("get" + name, new Class[]{});
  return m.invoke(agent, new Object[]{});
}
```

Ibis interceptors contact the Ibis parallelization services by creating instances of peer classes that are sent to an *Ibis network* for execution. The next subsection explains the anatomy of Ibis networks and how these interceptors interact with them.

6.2.4.2 The Ibis Server

Deploying and running a pure Ibis parallel application requires to carry out a number of manual configuration steps. First, the application bytecode has to be copied to the Grid hosts that will participate in the execution of the application. Second, each node must be explicitly assigned a numeric global identifier (which ranges from zero to the number of hosts participating in the run less one), the unique identifier of the application, and the address and port of the so--called *nameserver* (usually one of these machines). Nameservers provide runtime information about a particular run, such as determining the applications that are being executed, finding

hosts participating in a run, providing address and port information, and so on. Finally, after all these steps have been accomplished, the application is launched by manually initiating it in every host. Hosts coordinate themselves to cooperatively and efficiently execute the Ibis application.

Nonetheless, the above mechanism has some clear drawbacks. On one hand, it is literally too manual, as it demands the user to be excessively involved in the deployment, configuration and even the execution of applications. On the other hand, the mechanism is very inflexible, since the application code that is executed on the Ibis platform is determined statically. After launching, applications execute their associated main method that invokes the actual divide and conquer spawnable code, and then die. This is to say, it is not possible to dynamically parametrize the Ibis parallelization module with the code to be executed.

To alleviate these problems, JGRIM provides the `IbisServer` class. Basically, this class implements a pure Ibis application –that is, it is compliant to the Ibis application structure and its bytecode is properly instrumented through the Ibis compiler– that is able to execute other Ibis applications. An Ibis network is statically established by simultaneously configuring and starting the Ibis server on one or more hosts. A network is identified by the port on which it listens for incoming application execution requests. A request comprises three elements: a method signature, invocation arguments, and a target Java object (e.g. a peer) that represents an instance of the Ibis application on which the method must be executed. Instances of these Java objects specialize the behavior of the `SatinObject` Ibis API class. To send a `SatinObject` for execution on an specific Ibis network and wait for the results, the `IbisClient` is used.

Each JGRIM host may or may not be part of an Ibis network. In case it does, the host is configured to point to a specific Ibis nameserver, which in this context represents an arrangement of machines running the Ibis server application. Moreover, an individual host can belong to one or more Ibis networks, playing the role of either a slave or a master (i.e. nameserver) machine within a single network. The parameters that must be supplied to configure a JGRIM host as a node of an Ibis network are the IP address and the port to which the network's nameserver is bound.

Furthermore, all JGRIM hosts are statically supplied with the address of the Ibis machine that is contacted by JGRIM execution interceptors to run Ibis applications, and the specific port (*execPort*) where the Ibis server application running in that network is listening. Basically, the entry point to an Ibis network is the Ibis instance listening on `nameserver:execPort`. Figure 6.8 exemplifies a Grid that is composed of a number of JGRIM and Ibis-enabled machines.

As depicted, a logical JGRIM network including two Ibis networks A (with hosts H_1, H_2 and H_3) and B (with hosts H_3 and H_4) have been configured. The Ibis nameserver of A and B are hosts H_1 and H_4, respectively. In consequence, two instances of the Ibis server will be run, waiting for incoming execution requests on $(H_1, 10000)$ and $(H_4, 10000)$. In this way, spawns generated by applications received through the former/latter endpoint will be executed on the machines of the Ibis network A/B. By default, Ibis interceptors located at a host H_i send all execution requests to the nameserver of the network to which H_i belongs. In case H_i is part of

more than one network (e.g. H_4), the nameserver that is to be used must be chosen at Grid deployment time.

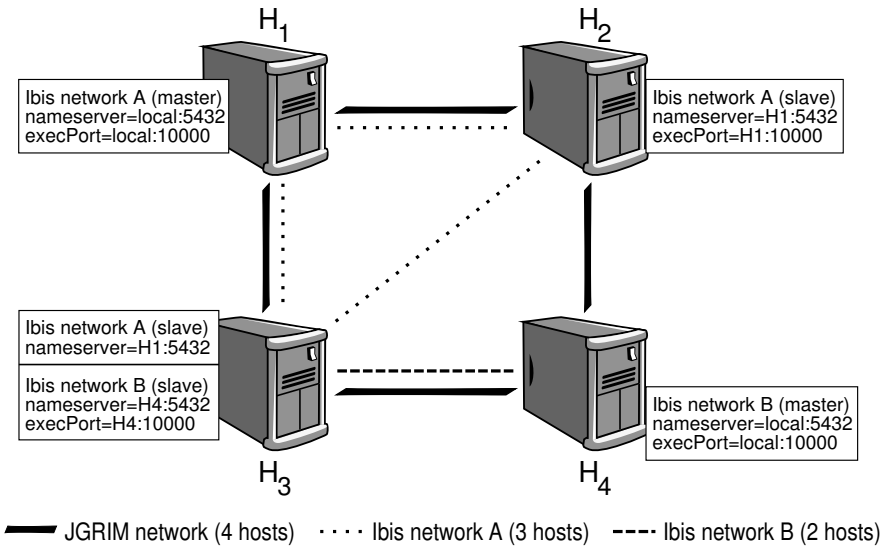


Figure 6.8: JGRIM hosts and Ibis networks

To minimize WAN and Internet traffic, it is usually a good idea that most of the hosts of a JGRIM network be configured to point to a LAN-accessible Ibis entry point. For instance, the Grid setting that were used to evaluate JGRIM (described in the next chapter) was composed of a JGRIM network across three Internet-connected clusters, and three Ibis networks, each one having a nameserver located on a different cluster. Additionally, hosts within a particular cluster were configured to access its local nameserver machine as the Ibis entry point for the execution self-dependency methods.

Figure 6.9 shows the JGRIM components that are involved in the execution of the self-dependency method of the example discussed throughout this section. When the MGS invokes its `methodB` operation, the injected interceptor creates an instance of the `IbisClient` class based on the Ibis entry point configured for the host where the MGS is currently executing. Then, an instance of its associated peer is built by injecting to it the both the MGS instance variables not related with dependencies and a "flattened" version of the agent. Finally, the peer and the information for executing `methodB` is sent to the Ibis server application. Eventually, the computation finishes and the Ibis server delivers the result back to the interceptor, which in turn passes it to the application.

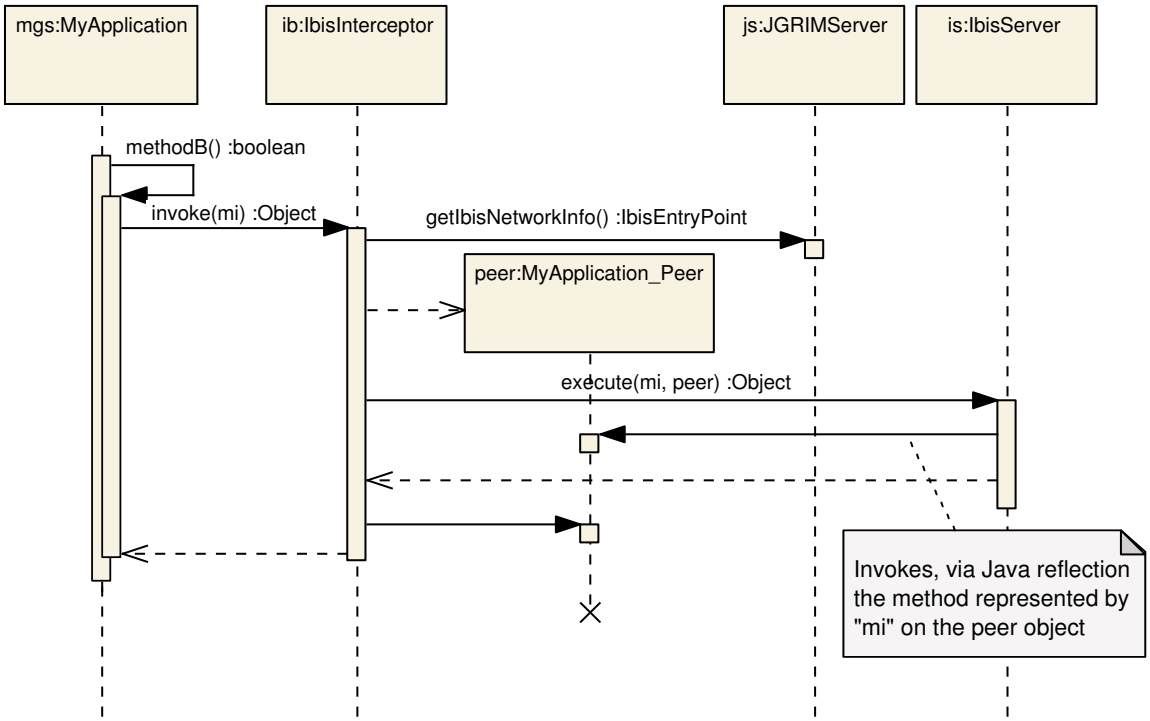


Figure 6.9: Execution of self-dependency methods as Ibis applications

6.3 Conclusions

The JGRIM middleware is implementation of the GRATIS approach discussed in Chapter 4, which also materializes the concepts of the GRIM model. Particularly, the present chapter discussed in detail the aspects related to the design and implementation of JGRIM. The middleware is purely based on Java, which makes it portable to any operating system for which a JVM implementation exists. In addition, JGRIM is designed to be extensible, and it is fully integrated with Web technologies, thus it can be used not only for building Grid settings but also for serving as a platform on which other types of distributed environments and applications can be built. Examples are mobile agent systems, service-oriented software and e-commerce applications.

In other respect, JGRIM served as a mean of assessing the benefits of the approach described in this thesis. Specifically, several experiments using JGRIM and some of the related alternatives were conducted on a real Grid setting to corroborate the practical soundness of GRATIS. Comparison was performed based on common dimensions of gridification such as effort, flexibility and performance. The next chapter describes these experiments.

Chapter 7

Experimental Results

In previous chapters, a novel approach to gridification called GRATIS and a concrete implementation for it named JGRIM were presented. This chapter describes the experimental evaluation that was carried out in order to provide empirical evidence about the practical soundness of the GRATIS approach.

In short, the Ibis, ProActive and JGRIM middlewares were separately employed to gridify existing implementations of two different applications, namely the k-NN clustering algorithm (discussed in Section 4.3.4), and an application for enhancement of panoramic pictures based on the restoration algorithm proposed in (Tschumperlé and Deriche, 2003). After gridification, code metrics on the Grid-aware applications were taken to quantitatively analyze how hard is to port a Java application to a Grid with either of the three alternatives. In addition, experiments were conducted to test the various performance aspects of JGRIM applications with respect to related approaches. To this end, a real Grid setting deployed over a WAN was used. The numerical data obtained from the experiments can be found in Appendix C (k-NN algorithm) and Appendix D (image restoration). The code implementing the different variants of the k-NN and restoration applications is shown in Appendixes A and B.

The chapter is organized as follows. The next section describes the Grid setting that was configured to perform the above experiments. Results obtained from the experimentation with the k-NN algorithm and the picture restoration application are described in Sections 7.2 and 7.3, respectively. Finally, in Section 7.4, the conclusions of the chapter are presented.

7.1 The Grid Setting

Figure 7.1 shows the topology of the Grid setting that was used to run the experiments. Specifically, three clusters named ISISTAN, Bianca and Ale, each composed of a certain number of machines, were linked through a Virtual Private Network (VPN) by using OpenVPN (Feilner, 2006). OpenVPN is an open source software package for creating point-to-point encrypted tunnels between Internet-connected computers. It is worth mentioning that the machine "gw_isistan"

in the figure just represents the network infrastructure of the ISISTAN Research Institute at UNICEN, but it did not participate in the runs. Moreover, machines of the ISISTAN cluster are part of a larger, public network that shares the 2 MB ADSL link. In contrast, the other two clusters are basically local private networks with exclusive access to their associated Internet link.

An OpenVPN server (version 2.0.9) was installed on the "VPNServer" machine of the ISISTAN cluster, thus establishing a simple star configuration for the resulting virtual private network. On average, communication between nodes of the Bianca or the Ale cluster and any machine of the ISISTAN cluster experienced a latency in the range of 60-90 milliseconds, whereas communication between Bianca and Ale clusters was subject to a latency of 100-150 milliseconds. All test described further in this chapter were performed during night time (approximately from 11 P.M. to 8 A.M.), that is, when the Internet traffic is low and network latency has little variability. Furthermore, all test batteries were launched from the machine named "ale".

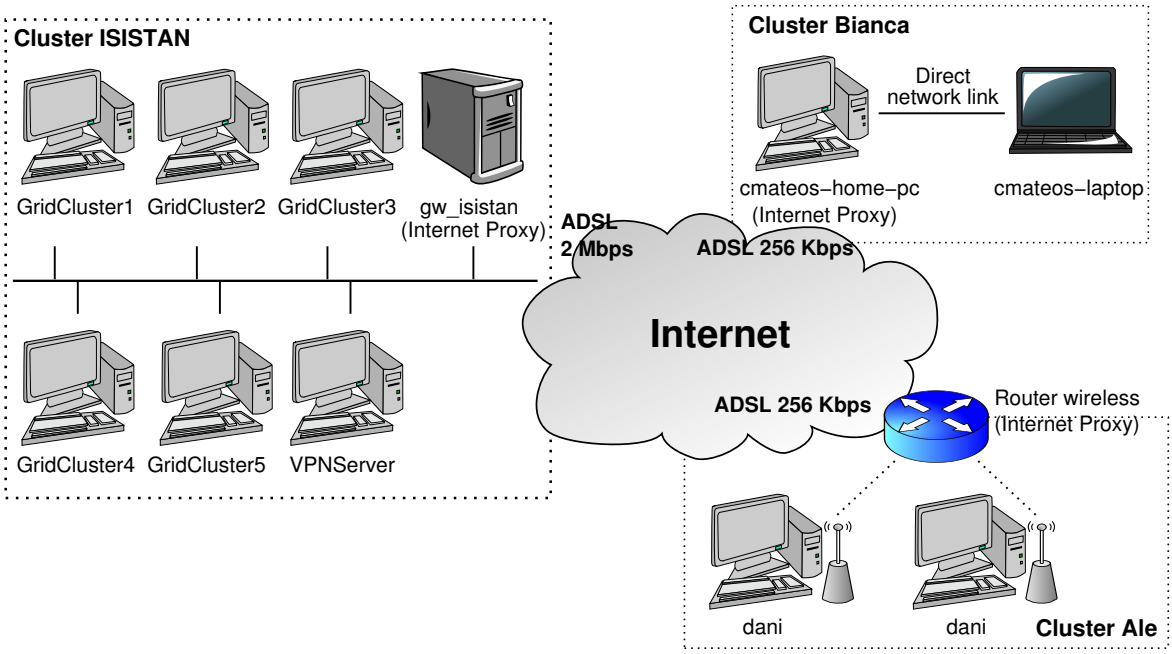


Figure 7.1: Network topology used for the experiments

Table 7.1 shows the CPU[1] and memory[2] specifications for the various machines of the Grid setting previously shown. All machines were equipped with either Kubuntu Linux or its lightweight version Xubuntu, both running Ubuntu kernel version 2.6.20. Besides, the Sun JDK version 1.5.0 was used. Finally, to keep the system clocks accurately synchronized across the Grid, a Network Time Protocol (NTP) [3] daemon was installed on every machine. NTP is a protocol for synchro-

[1]Dual core feature was disabled to alleviate the differences in processing capabilities
[2]As reported by the *top* linux command
[3]The Network Time Protocol project: ttp://www.ntp.org

nizing the clocks of computer systems over packet-switched networks that is designed mainly to resist the effects of variable latency or *jitter*.

As the reader can see from the table, both CPU power and memory availability significantly varies not only between different clusters, but also among machines of the same cluster. Similarly, bandwidth and latency across cluster machines are quite different. However, note that this is not a problem but a distinctive feature of Grid-like environments, which are indeed characterized by being an arrangement of (usually very) heterogeneous machines connected through network infrastructures with different capabilities.

Machine name	CPU model	CPU frequency	Memory (Kb)
GridCluster1	Pentium III (Coppermine)	852 Mhz.	247.960
GridCluster2	Pentium III (Coppermine)	852 Mhz.	247.960
GridCluster3	Pentium III (Coppermine)	852 Mhz.	377.812
GridCluster4	Pentium III (Coppermine)	852 Mhz.	377.812
GridCluster5	Pentium III (Coppermine)	798 Mhz.	256.160
VPNServer	Intel Pentium 4	2.80 Ghz.	516.040
cmateos-home-pc	AMD Athlon XP 2200+	1.75 Ghz.	256.092
cmateos-laptop	Intel Core2 T5600	1.83 Ghz. (per core)	1.027.528
ale	AMD Athlon 64 X2 Dual Core 3.600+	2 Ghz. (per core)	904.660
dani	AMD Sempron	1.9 Ghz.	483.472

Table 7.1: CPU and memory specifications of the Grid machines

7.2 The k-NN Clustering Algorithm

The k-nearest neighbor (k-NN for short) (Dasarathy, 1991) is a supervised learning technique where an object instance is classified by being assigned the most common class label among its k nearest neighbors in a multidimensional feature space. In other words, given a new instance (or query point), the algorithm finds k objects (or training instances) closest to the query point.

Suppose we have a dataset containing data about paper tissues, where each instance is represented by two attributes (acid durability and strength) and a class value indicating whether a particular paper tissue is good or not. Additionally, let us suppose that the class label associated to each instance has been assigned by collecting the opinion of many people. Table 7.2 shows four training samples of this hypothetical dataset. Now, if a factory produces a new paper tissue that pass laboratory test with acid durability = 3 and strength = 7, the k-NN algorithm can

predict the quality of the paper without the need of conducting another (possibly expensive) survey.

Acid durability (seconds)	**Strenght (kg/square meter)**	**Class (quality)**
7	7	Bad
7	4	Bad
3	4	Good
1	4	Good

Table 7.2: A sample dataset with four training instances

The k parameter of the k-NN algorithm is a positive integer, typically small. The best choice of k depends upon the data. Usually, larger values of k reduce the effect of noise on the classification, but make boundaries between classes less distinct. A good k can be selected by various heuristic techniques (e.g. cross-validation). To experiment with long-running, time-consuming tests, the performance evaluation of this algorithm described later in this section used a fixed, large value for k.

Algorithm 1 The k-nearest neighbor algorithm

```
procedure CLASSIFY(instance, k)        ▷ Returns the class label associated to an instance
    double[] attrs ← GETATTRIBUTES(instance)
    Vector neighbors ← INITIALIZENEIGHBORSLIST(k)
    for all trainingInstance ∈ Dataset do
        double[] trainingAttrs ← GETATTRIBUTES(trainingInstance)
        String trainingClassLabel ← GETCLASSLABEL(trainingInstance)
        double distance ← EUCLIDEANDISTANCE(instance, trainingInstance)
        SORTEDINSERTION(neighbors,distance,trainingClassLabel)        ▷ neighbors is kept
sorted (smaller distances first)
        if LISTSIZE(neighbors) > k then
            REMOVELAST(neighbors)
        end if
    end for
    return MOSTFREQUENTLABEL(neighbors)
end procedure
```

A pseudo code for the k-NN is shown in Algorithm 1. The original version (i.e. non-Grid aware) of this algorithm was implemented as a single Java class accessing a file-based dataset through a `Dataset` class. The application provided two operations `classifyInstance` and `classifyInstances` for classifying and individual instance and a list of instances, respectively. On the other hand, `Dataset` included a method for reading a block of training instances, and methods for obtaining the size and the number of dimensions of the dataset.

In order to establish a simple service-oriented Grid environment, the dataset was wrapped with a Web Service exposing analogous operations to those implemented by the `Dataset` class, and a replica of this service –along with its associated data– was deployed on each cluster. The original dataset had a size of 10.000 records, each described by 20 attributes with randomly-generated numerical values, and a numerical class label taking its value at random from a set of three predefined categories. The dataset was built by using the Weka[4] data mining toolkit. Finally, a UDDI registry pointing to the WSDL definitions of the service replicas was installed on the ISISTAN cluster.

Sections 7.2.1 and 7.2.2 present the experimental results obtained from the gridification of the Java implementation of the k-NN algorithm previously mentioned. Specifically, the former section describes the gridification effort incurred by each Grid platform, whereas the latter concentrates on evaluating performance issues.

7.2.1 Gridification Effort

The purpose of the analysis presented in this section is to measure the effort to gridify the existing k-NN implementation, and the impact of this process on the application design and implementation. Certainly, quantifying these aspects is very difficult. Therefore, the analysis focuses on elaborating and then applying an formula to quantitatively estimate the effort required when using Ibis, ProActive and JGRIM. The resulting formula was also employed to quantify the gridification effort of the restoration application of Section 7.3.

Essentially, the estimation of the effort invested in gridification was performed by comparing the values of relevant code metrics for both the original application and its gridified counterparts. Compilation units being part of the dataset Web Service implementation were not considered by the estimation, because the experiments were carried out as if the Grid (and therefore its services) was already established and performing gridification thereafter. The following list summarizes the code metrics that were employed:

- *TLOC (Total Lines Of Code)*: It counts the total non-blank and non-commented lines across the entire application code, including the code implementing the algorithm itself, plus the code (when applicable) for interacting with the dataset, performing Grid exception handling and taking advantage of execution parallelization. This metric is directly related to the extra implementation effort that is necessary to prepare the source code of an ordinary application to execute on a Grid platform.

- *PLOC (Platform-specific Lines Of Code)*: This metric counts the number of source code lines that access the underlying API of the target Grid platform. Specifically, instructions using API classes or invoking methods defined in these classes are computed as a PLOC line.

[4]Weka version 3: http://www.cs.waikato.ac.nz/ml/weka

Note that the larger the value of PLOC, the more the level of tying between the application code and the Grid platform API. Clearly, it is highly desirable to keep PLOC as low as possible, so as to avoid applications to be dependent on a particular Grid platform, which in turn hinders their portability to other platforms. Intuitively, as PLOC grows, so does the time that must be spent learning the corresponding platform API, since the probability of using different portions of this API increases.

- *NOC (Number Of Classes) and NOI (Number Of Interfaces)*: They represent the number of implemented application classes and interfaces, respectively, without taking into account classes or interfaces provided by the underlying Grid platform API, the Java API or third-party libraries. Although simple, note that these metrics are useful to give an idea of the amount of object-oriented design present in the application.

- *NOT (Number Of Types)*: It simply counts the number of object types (classes and interfaces) which are defined in, and referenced from within, any of the compilation units of the application. The NOT metric does not consider neither the Java primitive types nor the JVM bootstrap classes defined in the *java.lang* package. NOT can be viewed as the sum of NOC, NOI and the number of classes/interfaces used after the Java reserved keywords *extends*, *implements* or *import*. As a corollary, the formula $NOT - (NOC + NOI)$ yields as a result the number of classes/interfaces not considered by the NOC/NOI metrics, that is, the object types which are defined in either the runtime system API or external libraries. It is worth noting that a class which is simultaneously subclassed and imported –or similarly, an interface which is implemented and imported– is counted as *one* object type.

Table 7.3 shows the resulting metrics for each one of the four implementations of the k-NN application: original, Ibis, ProActive and JGRIM. Figure 7.2 summarizes the measured TLOC, and Figure 7.3 shows the overhead incurred in gridifying the application in terms of source code lines. In order to perform a fair comparison, the following tasks were carried out on the implementation code:

- The source code was transformed to a common formatting standard, thus sentence layout was uniform across the different implementations of the application.

- Java import statements within compilation units were optimized by using the source code optimizing tool of the Eclipse IDE. Basically, this tool provides support for automatic import resolution, thus leaving in the application code only those classes/interfaces which are actually referenced by the application.

- Applications were Java 1.5 compliant, but Java generics across the source code were removed. The goal of this task was to avoid counting a line including a declaration of the form <*PlatformClass*> as a PLOC line. Otherwise, variants repeatedly using this feature across the code (e.g. in method signatures, data structure declarations, etc.) would have been unfairly resulted in greater PLOC.

Besides, all Grid-aware versions were implemented by the same person. The developer had very good expertise on distributed programming, and a minimal background on the facilities provided by either of the three Grid platforms. In this way, the analysis is not biased by the experience, or by different design and implementation criteria that potentially might have arisen if more than one person were involved in the gridification of the application.

k-NN version	TLOC	PLOC	NOC	NOI	NOT	Code overhead (%)
Original	192	–	4	0	11	–
Ibis	1477	10	25	3	79	669.27
ProActive	404	11	5	**0**	37	110.42
JGRIM	**166**	**4**	**4**	2	**12**	**-13.54**

Table 7.3: Gridification of the k-NN algorithm: code metrics

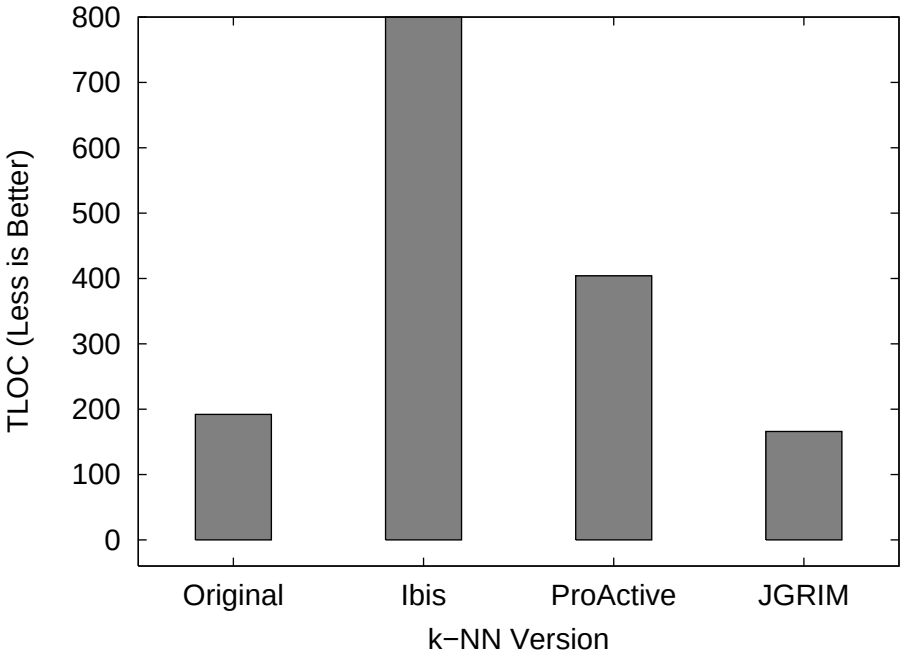


Figure 7.2: TLOC after gridification

7.2.1.1 Ibis

The size of the application in the case of Ibis was 1477 lines (seven times bigger than the original implementation). As a consequence, only a small percentage of the code resulted in

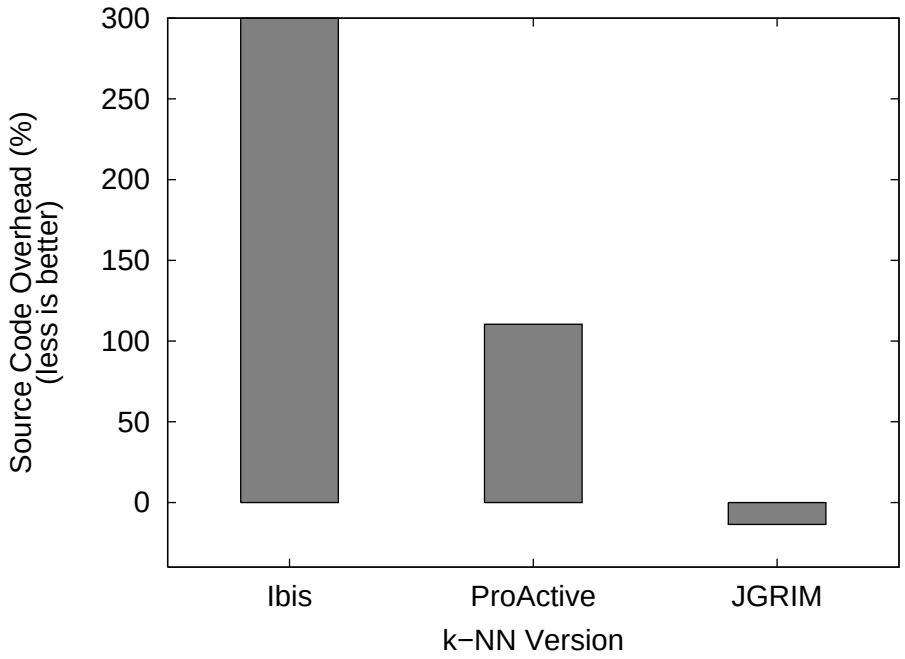


Figure 7.3: Source code overhead introduced by the gridification process

pure application logic, since it was necessary to provide a lot of code mostly to generate and use a client-side proxy to the dataset Web Service. Therefore, NOC and NOT also suffered (six and seven times bigger, respectively), since more application classes and interfaces were created, and also extra APIs for low-level interaction with Web Services were imported. Surprisingly, despite being a platform proposed for Grid development, Ibis does not offer any facilities for using Web Services.

As the original application was thought to be executed sequentially on a single machine, `classifyInstances` (see Figure 7.4) was straightforwardly implemented by means of a loop control structure that iteratively feeds the `classifyInstance` method with the items of the list received as an argument. To take advantage of Ibis parallelism, `classifyInstances` was rewritten to use a spawnable classification method (`spawnclassify`) which in turn directly calls `classifyInstance`. Hence, at runtime, each invocation to `classifyInstance` is concurrently executed by the Ibis platform. In this way, significant performance benefits were obtained at the cost of having an extra implementation effort since more changes to the original application were introduced. Figure 7.4 shows a simplified class diagram of the resulting application. Communication and coordination between `classifyInstances` and spawned computations was achieved by means of a *shared object* (Wrzesinska et al., 2006), a mechanism provided by Ibis to transparently share and update the state of a Java object among the distributed spawned computations of an executing application.

To further optimize the application, the `IbisDatasetClient` class was implemented to choose, upon classification of a particular instance, the service replica that is located at the cluster where the associated spawned computation is executing. The dataset client operates as its own service

broker: when data needs to be read, the client simply performs a "ping" operation to select – based on communication latency– the most appropriate WSDL description from a list of known candidate addresses. Consequently, fast interaction with the dataset is achieved. However, this mechanism forced the code to be improperly tied to specific service instances, therefore lacking reusability as the information for invoking services on a Grid (e.g. WSDL location) or the available instances for a service may vary over time. Even when this problem can be solved by using a service registry, or alleviated by passing the available service descriptions as an argument to the application, the heuristic used for service selection still remains hardwired in the dataset client code. In consequence, using other selection heuristic requires to reimplement this code.

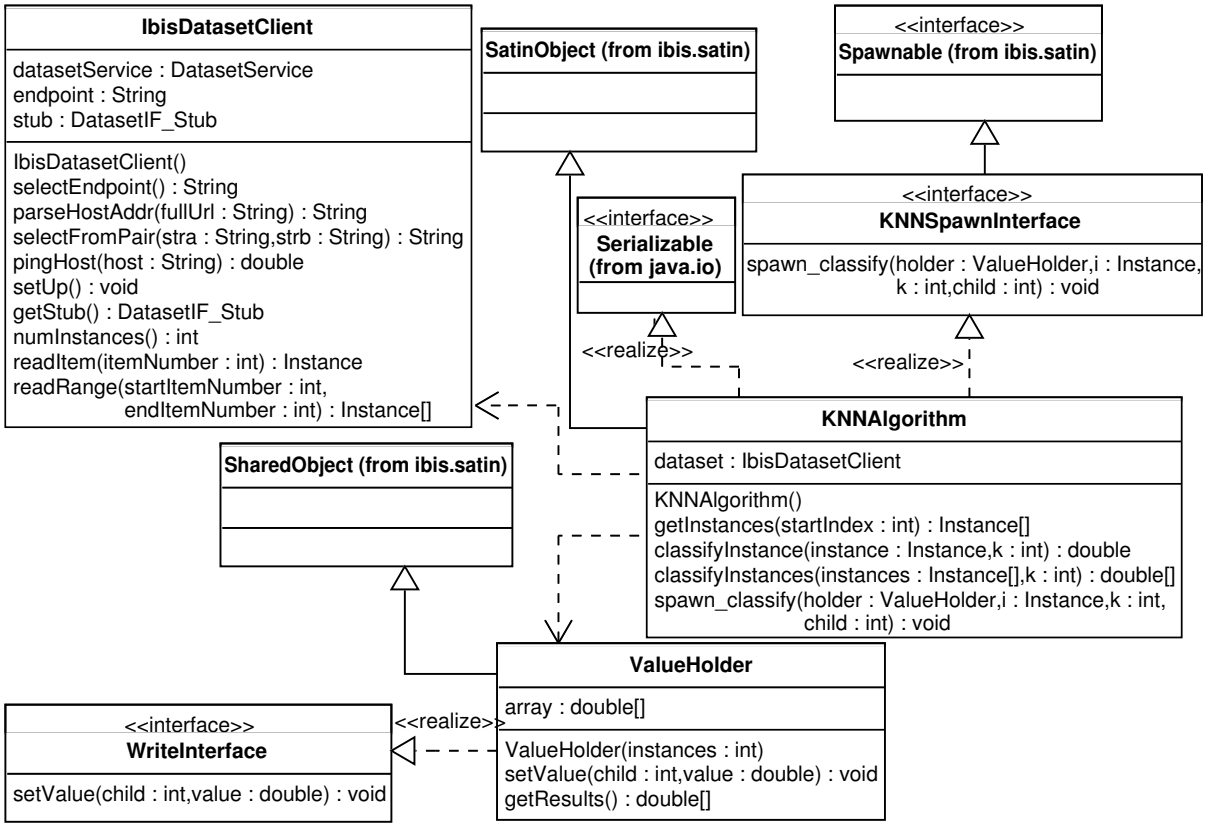


Figure 7.4: Class diagram of the Ibis version of the k-NN application

7.2.1.2 ProActive

The ProActive version of the application introduced a source code overhead near to 110%. The PLOC was slightly greater than for the Ibis case (11 lines, less than 3%) but, as indicated by NOT, the number of object types were reduced to less than a half (37 against 79). Figure 7.5 shows the class diagram of this implementation. Note that the class design is evidently much simpler than that of Ibis. However, several problems arised when gridifying with ProActive, which are described next.

ProActive also lacks full support for using Web Services, as it only offers a set of classes that allow applications to interact with either SOAP-enabled services or ProActive active objects. Web Service consumption within a client application is carried out by working directly with the SOAP APIs, since ProActive does not provide abstractions to transparently support other bindings to services such as CORBA or EJB. In this sense, since all dataset replicas were initially wrapped with a non-SOAP Web Service, it was necessary to implement an active object for interfacing the data, expose it as a SOAP service, and finally write a client to use it. Directly using SOAP instead of a more generic support for Web Service invocation significantly reduced both NOC and NOT, but caused the application to be tied to a specific transport protocol for interacting with services. Furthermore, in order to allow efficient interaction between the application and the dataset, `ProActiveDatasetClient` was instructed to employ the latency-based replica selection heuristic described before. In consequence, the ProActive dataset proxy shares the reusability and flexibility problems suffered by its Ibis counterpart.

Similar to Ibis, parallelization of `classifyInstances` was achieved by concurrently classifying individual instances at different Grid hosts. Specifically, a master-worker approach was followed in which, for each instance, a clone of the `KNNAlgorithm` class in the form of an active object is created, programmatically deployed on a particular host, and asked to perform a single classification. Whenever an active object becomes idle, another job is sent to it. Synchronization between the parent active object (i.e. the main execution thread of the application) and these clones was accomplished through the ProActive wait-by-necessity mechanism, which in this particular case was used to transparently block the execution of `classifyInstances` until any child active object finishes its assigned job.

Unfortunately, a significant amount of code to complement the ProActive synchronization support had to be supplied. The purpose of this code was mainly to implement behavior for keeping track of busy workers as well as assigning pending tasks to idle ones. Another alternative that was explored in an attempt to avoid this problem was to delegate job execution management to the platform. However, at the time the performance tests described later in this chapter were performed, load balancing in ProActive appeared to be a little unstable[5].

Finally, another negative aspect that arised as a consequence of parallelization was closely related to the ProActive wait-by-necessity mechanism. Basically, this mechanism forced the interface of the gridified application to differ from the interface of the original implementation, because a method must be changed its return type (i.e. replace it with some ProActive API class) to enable it to be invoked asynchronously. Particularly, the return type of the `classifyInstance` method was modified to return instances of the `GenericTypeWrapper` ProActive built-in object type. Consequently, the new application interface contained non-standard datatypes and interaction conventions. From a SOA point of view, this makes the interface of the ordinary version of the application no longer valid, as the gridified application does not adhere to the original service interface. Hence, application discovery within a Grid setting is more tedious and difficult, as

[5]The original post to the ProActive development community reporting this problem can be found in http://www.nabble.com/Deployment-and-load-balancing-t4126250.html

details on required service interfaces must be provided by external clients for the discovery process to be effective.

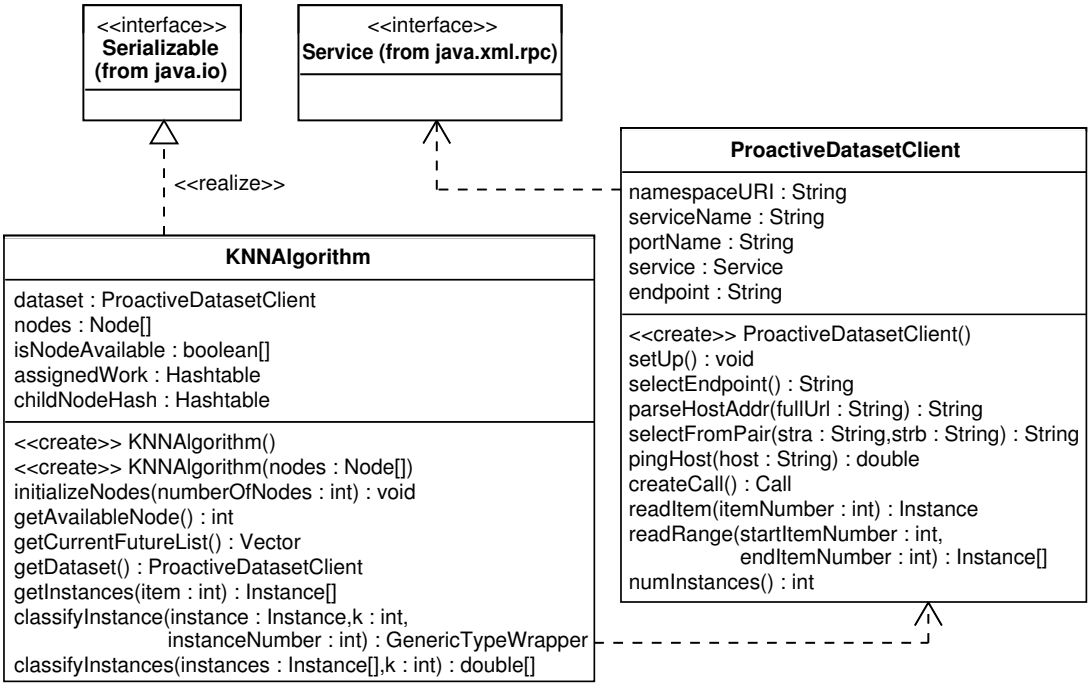


Figure 7.5: Class diagram of the ProActive version of the k-NN application

7.2.1.3 JGRIM

The gridification of the k-NN algorithm with JGRIM resulted in 166 lines of code, plus 126 lines of XML configuration automatically generated by the JGRIM gridification tool (similar values were obtained for the a variant using a policy, which is described later). The code was even smaller than the non-gridified version, since dataset access is transparently performed through the discovery and invocation metaservices. Besides, exceptions that may be thrown when calling services (e.g. communication errors, timeouts, etc.) are mostly handled at the platform level, which greatly helps in reducing the gridification effort.

The class diagram of the JGRIM k-NN application is illustrated in Figure 7.6. It is worth emphasizing that the tasks of extending the `JGRIMAgent` and `JGRIMPolicyAdapter` classes, adding proper setters/getters, and realizing both `DatasetInterface` and `ParallelMethodInterface` were performed automatically by JGRIM based on the dependencies declared at gridification time. From the diagram, it can be seen that the application logic (i.e. the `KNNAlgorithm` class) does not directly reference concrete components providing Grid behavior.

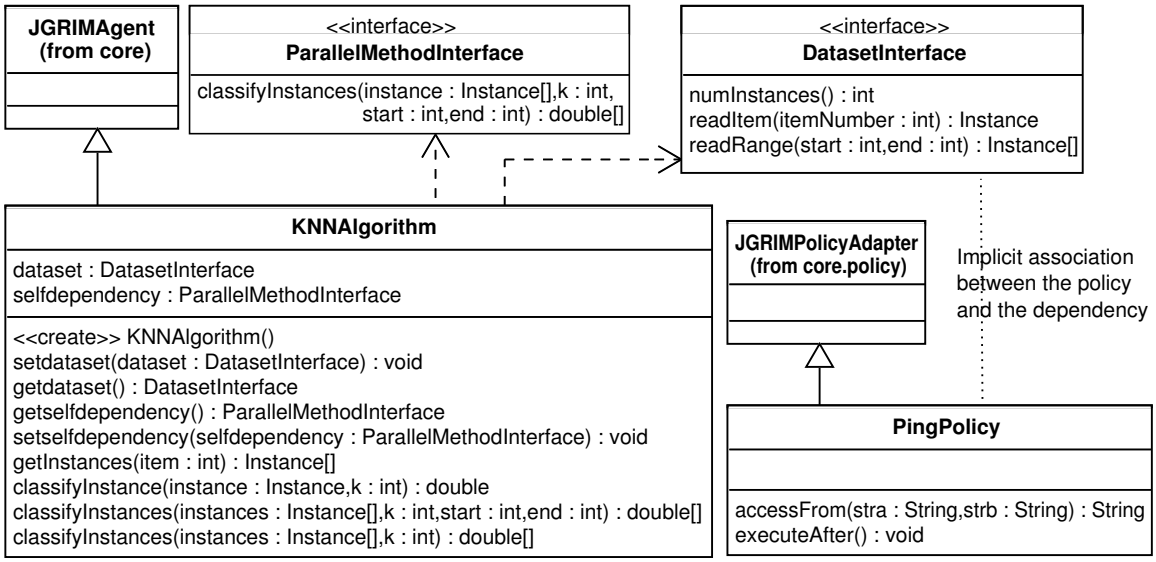


Figure 7.6: Class diagram of the JGRIM version of the k-NN application

The object decoupling and metaservice injection capabilities featured by JGRIM enabled to implement the interaction with the dataset with little coding effort. Only few lines (those invoking dataset operations) were altered to use the corresponding getter method, making the resulting code very clean and easy to understand. Moreover, introducing and testing further improvements over the algorithm outside the Grid setting is straightforward, since another source for the dataset dependency (e.g. a mock Java object) can be easily configured to the application without modifying its code. This is not always the case in Ibis and ProActive, since the portions of applications that are tied to Grid configuration information or technologies have to be rewritten or discarded.

To stay competitive, and performing a fair performance evaluation, a policy materializing exactly the same service selection heuristic used by the Ibis and ProActive implementations was configured for the JGRIM application. Apart from its benefits in terms of flexibility and reconfigurability, a very interesting aspect of the policy mechanism is that it concentrates the underlying platform details within a few classes that are external to the original application code. Specifically, besides using less API code than either Ibis or ProActive (4 lines against 10/11 lines) the totality of the PLOC lines of the JGRIM application were located exclusively in the classes implementing policies, which made the application logic free from platform API code. These lines were mostly calls to the profiling services provided by the JGRIM middleware.

The aspect of the JGRIM solution that demanded more attention from the developer was concerned with parallelization. As explained in previous chapters, JGRIM complements self-dependencies with a coordination technique that works by blocking the execution of an MGS the first time it requests the result of an unfinished spawned computation. As a consequence, the technique is by far less effective if an operation of a self-dependency is called inside a loop control structure which accesses the result of a call before another call takes place. For example, dumbly replacing "this" by a self-dependency in the original implementation of `classifyIn-`

stances would have resulted in a code similar to:

```
double[] classifyInstances(Instance[] instances, int k){
  double[] result = new double[instances.length];
  for (int i = 0; i < instances.length; ++i){
    iClass = getselfdependency().classifyInstance(instances[i], k);
    result[i] = iClass;
  }
  return result;
}
```

which clearly causes classification to be performed sequentially, as the computation of the class label associated to `instances[i+1]` do not start until the class value of `instances[i]` is available.

Parallelization of `classifyInstances` demanded to split its original implementation into two new operations: a method (keeping the original interface of `classifyInstances`) that accesses, through a self-dependency, another method implementing the actual classification process for a list of instances. Since the Ibis execution metaservice[6] was used to concurrently classifying instances, the latter method was implemented in a recursive way by following the Ibis coding conventions. The resulting code produced by this transformation was:

```
double[] classifyInstances(Instance[] instances, int k){
  return getselfdependency().classifyInstances(
                          instances, k, 0, instances.length);
}

double[] classifyInstances(Instance[] instances, int k,
                           int start, int end){
  if (end - start == 1){
    iClass = classifyInstance(instances[start], k);
    return new double[] {iClass};
  }
  int mid = (start + end) / 2;
  double[] res1 = classifyInstances(instances, k, start, mid);
  double[] res2 = classifyInstances(instances, k, mid, end);
  double[] result = new double[res1.length + res2.length];
  System.arraycopy(res1, 0, result, 0, res1.length);
  System.arraycopy(res2, 0, result, res1.length, res2.length);
  return result;
}
```

[6]Cluster-aware random stealing (CRS) was configured as the algorithm for job stealing

In this way, the execution of the second method is delegated to an external execution service (in this case Ibis), and the interface of the original `classifyInstances` method is maintained. Therefore, the resulting MGS exposes the same service interface as the original application. Additionally, note that the transformation did not require to use any Grid API code at all. Finally, many simple techniques can be found in the literature to deal with the old task of switching from iterative to recursive code (and viceversa).

The situation described before is a clear example of a common tradeoff likely to be found when parallelizing an application: independence from the parallel programming API versus flexibility to control the application execution (Freeh, 1996). From the programming language level, the approaches to parallel processing can be classified into implicit or explicit. On one hand, implicit parallelism allows programmers to write their programs without any concern about the exploitation of parallelism, which is instead automatically performed by the runtime system. On the other hand, languages based on explicit parallelism aim at supplying synchronization and coordination APIs for describing the way parallel computations take place. The programmer has absolute control over the parallel execution, thus it is feasible to deeply take advantage of parallelism to implement very efficient applications. However, programming with explicit parallelism is more difficult, since the burden of initiating, stopping and synchronizing parallel executions is placed on the programmer.

Platforms like ProActive and Ibis, which are inherently performance-oriented, are designed to provide explicit parallelization. The programmer has a finer control of parallelism, but gridified applications are rather difficult to understand and to maintain. Conversely, JGRIM promotes implicit parallelism, since it provides parallelization facilities in those places of an application where it can be transparently used, that is, without requiring to explicitly use extra API code. Unlike ProActive or Ibis, JGRIM pays attention not only to application performance, but also to portability, maintainability and legibility of the application code.

7.2.1.4 Discussion

An interesting result of the evaluation is concerned with the size of the runnable versions of the different k-NN implementations described in the above sections. Table 7.4 shows application bytecode information, given by the number of generated .class files and the total size (in Bytes) of these files after compilation, and plugging information, given by the number of total lines of code (and PLOC lines) destinated to run the application on the Grid.

After source code gridification and compilation, the binary size of the ProActive implementation was about 17 KB, versus 31 KB and 108 KB of the JGRIM and Ibis, respectively. In this latter case, though much functionality for interacting with the dataset was added to the original application (resulting in approximately 74 KB of bytecode), the final deployment generated a lot of .class files for performing parallelization, managing shared objects and carrying

k-NN version	# .class	Bytecode size	Bytecode overhead (%)	Test lines/PLOC
Original	4	8242	–	–
Ibis	45	111452	1252	43/0
ProActive	5	17750	115	72/15
JGRIM	13	32028	288	55/4

Table 7.4: Characteristics of the k-NN implementations upon execution on the Grid setting

out platform-specific object serialization, therefore increasing the amount of bytecode of the whole application. In the end, transferring the code for application execution on a remote host will require more bandwidth than either the ProActive or JGRIM implementations. For more complex applications, the extra required bandwidth could be bigger.

The bytecode of the ProActive solution was about a half of the binary code generated by the JGRIM implementation. However, it is worth noting that ProActive dynamically adds mobility to applications by enhancing their bytecode not at deployment time, but at runtime, thus this overhead is not present in the above results because it is very difficult to measure. On the other hand, it was determined that a big percentage of bytecode for the JGRIM implementation were instructions specifically targeted for dealing with thread-level serialization and migration, and specially creating the Ibis peer for the main application class, this latter representing a 72% of the total bytecode. In fact, the binary size without this support was even smaller than the compiled version of the original k-NN implementation. In this sense, in order to make the binary code lighter and more compact, at least in appearance, a technique for instrumenting bytecode similar to the one used by ProActive could be implemented. Basically, the idea is to develop a special Java class loader that instruments applications at runtime, thus dynamically enabling them for being parallel as well as mobile. In this way, dynamic code transfer is more efficient: when a host do not have the necessary bytecode to execute an application, the binary code that is transferred to it is just the non-instrumented version, thus saving network bandwidth.

Table 7.4 also includes the amount of source code that was implemented in order to allow applications to be tested and executed, and how much of this code was concerned with accessing the corresponding platform API. To a certain extent, the amount of implemented lines can be interpreted as an indicator of the effort demanded by either of those platforms to execute a gridified application onto the Grid. Quantifying this effort is important since a major goal of Grids is to be used to run user applications in a *plug and play* fashion. Taking into account that learning a Grid API is indeed a time-consuming task, this effort could be approximated by the following formula:

$$PlugEffort = (TestLines - PLOC_{Test}) + PLOC_{Test} * APIFactor$$

where $APIFactor$ is a numeric value that represents the complexity inherent to the platform API being used. Basically, the formula adjusts the lines representing Grid code to incorporate the effort invested by the programmer in learning the API. As here defined, $APIFactor$ is highly

influenced by how much does the application programmer know about a particular Grid API and its underlying abstractions. In this case, as the k-NN application was gridified by one person initially not having solid knowledge about any of the different Grid platforms, we can assume this influence is not present. In addition, since the developer had a very good background on distributed and parallel programming, the complexity associated to each API is very similar. Hence, we can naively assume that $APIFactor_{Ibis} = APIFactor_{ProActive} = APIFactor_{JGRIM}$ holds, which means he spent almost the same time to learn the three APIs.

Of course, the above formula is a very rough approximation to truly quantifying the necessary effort to execute a gridified application on the Grid. Certainly, many aspects intimately related to Grid application execution and deployment (e.g. creating/editing application configuration files, performing network-specific settings, initiating the execution of the application itself, and so forth) are obviously out of the scope of this formula. However, as the purpose of this thesis is not to address gridification beyond implementation code, the formula is a good approximation.

We can extend this idea to take into account the code metrics reported previously to obtain an estimation of the effort incurred by each middleware when gridifying an application. According to the taxonomies presented in Chapter 3, two fundamental aspects that characterize the existing gridification tools are concerned with how much redesign and code modification (within compilation units) they impose on the input application. Hence, the overall gridification effort can be thought as composed of three different effort factors: restructuring the application, adapting its individual compilation units, and plugging the resulting application into the Grid (modeled by $PlugEffort$).

Provided that the implementation language of the original application and the gridified application is the same (which is the case of the experiments of this chapter), we can obtain an estimation of the effort –in terms of source code lines– that is necessary to carry out this transformation, plus the effort to put the application to work, by the formula:

$$GE\,(Gridification\,Effort) = ReimplEffort + RedesignEffort + PlugEffort$$

where:

$$ReimplEffort = |TLOC_{Grid} - TLOC_{Orig}| + PLOC * APIFactor$$

$$PlugEffort = (TestLines - PLOC_{Test}) + PLOC_{Test} * APIFactor_{Test}$$

The reimplementation effort is computed as the difference between the amount of source code lines of the original and the gridified application, plus the adjustment of PLOC lines. If the difference is positive, extra lines to the original implementation has been added, whereas a negative difference indicates that the size of the new application is smaller. To model these cases with a single expression, the formula assumes that adding lines is as laborious as removing lines from the original application. As the redesign tasks performed on the original application were straightforward, $RedesignEffort$ was assumed to be almost zero. As a consequence, this factor was not considered in the computation of the values of GE for the three middlewares.

Intuitively, $APIFactor$ at implementation time should always be greater than the one included in the computation of $PlugEffort$, because at plugging time the developer is likely to be more

familiarized with the Grid programming API. In our experiments, this rule does not hold since the portions of the platform APIs that were used to perform both steps (i.e. object types) were almost disjunct. Therefore, the $APIFactor$ at implementation and plugging time are the same, and they were both set to 30 (i.e. a PLOC line is worth 30 regular or non-PLOC lines).

Since the gridification processes of Ibis, ProActive and JGRIM produce Java applications, comparison of the different GE values resulted from gridifying the original k-NN application –which was also written in Java– is absolutely feasible. To further simplify the comparison and to better perceive the relative differences between the computed values, GE was normalized according to the following formula:

$$NormalizedGE = \frac{GE}{ScalingFactor}$$

where:

$$ScalingFactor = 10^{truncate(log_{10}[max(GE_{Ibis}, GE_{ProActive}, GE_{JGRIM})])}$$

In the end, applying the GE indicator to the gridified versions of the k-NN algorithm resulted in 1.63 (Ibis), 1.05 (ProActive) and 0.41 (JGRIM) (see Figure 7.7). Roughly speaking, GE suggests that Ibis was the middleware demanding the greatest amount of effort from the developer. Furthermore, ProActive and JGRIM decreased this effort by 36% and 75% respectively, which means that the smallest effort was achieved by using JGRIM. Once again, it is worth emphasizing that these results cannot be generalised, since they merely represent an indicator of how much effort each middleware demanded to gridify the original k-NN implementation taking into special account the assumptions formulated in previous paragraphs.

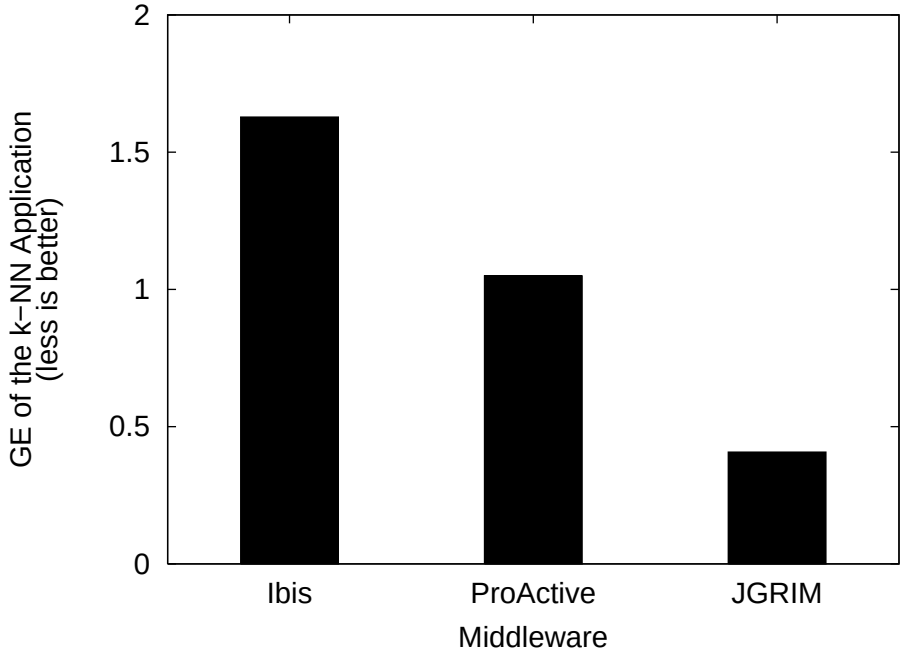


Figure 7.7: Gridification effort for the k-NN application

7.2.2 Performance Analysis

In order to evaluate the runtime behavior of the applications with respect to response time and network resource usage, each gridified version of the algorithm was employed to classify several list of instances with different sizes. Network resource consumption –specifically, the generated TCP traffic and the amount of data packets– were measured through the *tcpdump* network monitoring program[7] and then analyzed with the help of the Wireshark[8] software tool. It is worth mentioning that loopback network traffic was filtered out.

Furthermore, all tests were initiated in one machine ("ale"), thus launching conditions were exactly the same. In addition, since Ibis does not support dynamic transfer of application bytecode, the executable codes of all the gridified applications were manually copied to each host of the Grid. This task, which contributed to make a fair estimation of the network resources used by each application, was also carried out before experimenting with the restoration application described later in this chapter.

Each test battery performed on a gridified application involved ten executions of the classification algorithm on input lists composed of 5, 10, 15, 20 and 25 instances. The resulting execution times were averaged to compute the total execution time (TET). On the other hand, for the sake of simplicity, network resource consumption was measured by taking into account the total amount of network traffic and packets generated during an entire test battery.

7.2.2.1 Comparison of TET

The TET obtained from the different Grid-aware implementations of the k-NN algorithm are shown in Figure 7.8. Particularly, the figure depicts the average response time associated to the Ibis, ProActive and JGRIM implementations, and a variant of JGRIM using a policy that stores in an in-memory cache the totality of the data that is read from the dataset service. In addition, cache entries were configured to be non-volatile, thus cached information persisted across individual executions. As a consequence, dataset replicas were accessed completely by Grid hosts only once per test battery. The implementation associated to the policy was straightforward (13 lines of code), and it made explicit use of only two operations of the JGRIM policy API. The figure also shows errorbars in the y axis corresponding to the standard deviation in each case.

The ProActive application experienced low performance levels with respect to those achieved by the Ibis application or even the plain version of JGRIM (i.e. not using the caching policy). A great percentage of the time was spent by ProActive in remotely creating JVMs on every Grid node, which demanded on average 40 seconds. Easy deployment of Grid applications is

[7] Tcpdump/libpcap: http://www.tcpdump.org
[8] The Wireshark Network Protocol Analyzer: http://www.wireshark.org

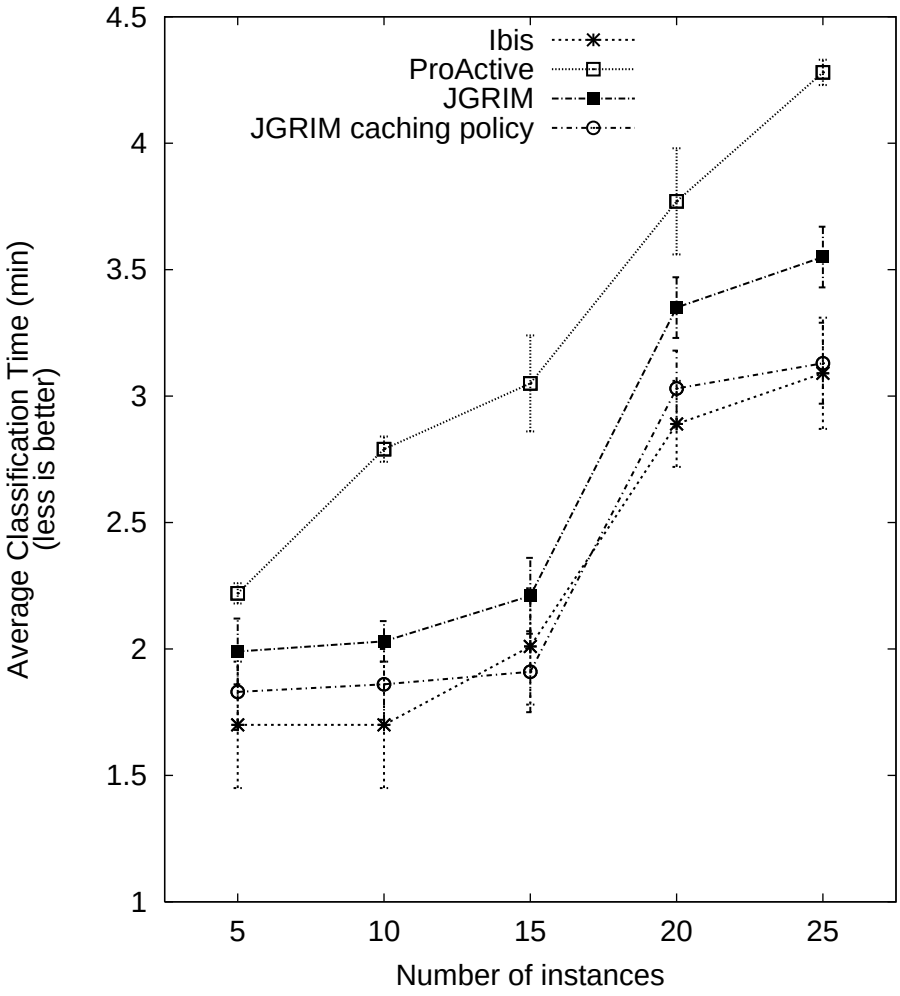


Figure 7.8: TET (min) of the k-NN application

indeed one of the good features of the ProActive middleware. However, the results analyzed here show that this feature directly conditioned the overall performance of the gridified application. In principle, the results suggest that ProActive may not be suitable for running computations whose response time is similar or just a few factors greater than the time required to remotely initialize the ProActive runtime system.

The Ibis application performed clearly better than the JGRIM version. Specifically, the JGRIM added a performance overhead in the range of 10-20%. However, much of this time (20 seconds on average) corresponded to access to the UDDI registry. Despite the obvious overhead service brokering has associated, it allows the application code to be completely isolated from the actual service instances implementing a certain functionality, which are selected at runtime instead by the underlying middleware. In consequence, the implementation code is shorter and cleaner, and remains free from instructions for searching and configuring Grid services. This fact became evident when the TLOC of the Ibis and JGRIM implementations were analyzed. Furthermore, brokering enables for a better reuse of Grid services, since applications and services are bound in a dynamic, more flexible way. In other words, brokering allows executing applications to potentially discover new services as they are published. It is worth mentioning that brokering capabilities in JGRIM can be easily disabled without modifying an application by simply editing the configuration associated to the gridified application. In this way, the performance of the application increases, at the potential cost of underexploiting the available Grid services.

To keep the JGRIM application using brokering services (i.e. access the UDDI registry), and at the same time decrease its response time, a variant of the JGRIM application based on a simple caching mechanism was implemented. As depicted, the alternative version reduced the TET of the non-cached JGRIM implementation by 8-16%, and achieved performance levels similar to the Ibis implementation. Moreover, implementing the policy demanded some programming effort, and resulted in a code similar to:

```
public class DatasetCachePolicy extends PingPolicy {
  public Object handleInvocation() {
    int dataBlock = (Integer) getCallInfo().getArguments()[0];
    if (cacheHit(dataBlock))
      return getFromCache(dataBlock);
    return null;
  }
  public void executeAfter() {
    super.executeAfter();
    int dataBlock = (Integer) getCallInfo().getArguments()[0];
    putInCache(dataBlock, getCallResult());
  }
}
```

Basically, the policy was implemented as an extension of the ping-based mechanism for selecting the nearest dataset service replica mentioned previously (represented by the `PingPolicy` class). The code within the `handleInvocation` method is executed by JGRIM prior to obtain a block of instance data from the dataset service. As discussed in Section 6.2.2, if a non-null value is returned (a cache hit), the actual invocation to the dataset service do not proceed. Conversely, method `executeAfter` is executed by the JGRIM runtime right after an actual access to the service takes place. Therefore, code has been provided to cache the result of a call to the dataset service, which is accessed by means of the `getCallResult` API method.

The weak point of the solution is that it increased the total memory usage within the Grid by about 100 MB, that is, the extra memory that was allocated at every host to store Java objects representing the instances of the dataset. In production Grids, where many different applications compete for the available resources, allocating memory and storage resources at will may be disallowed. However, the goal here was to show the flexibility of policies to provide user code to effectively and non-invasively tune JGRIM applications. Note that both the ProActive and Ibis applications might have benefited from the same caching mechanism, but this would have forced to introduce yet more modifications to the original application.

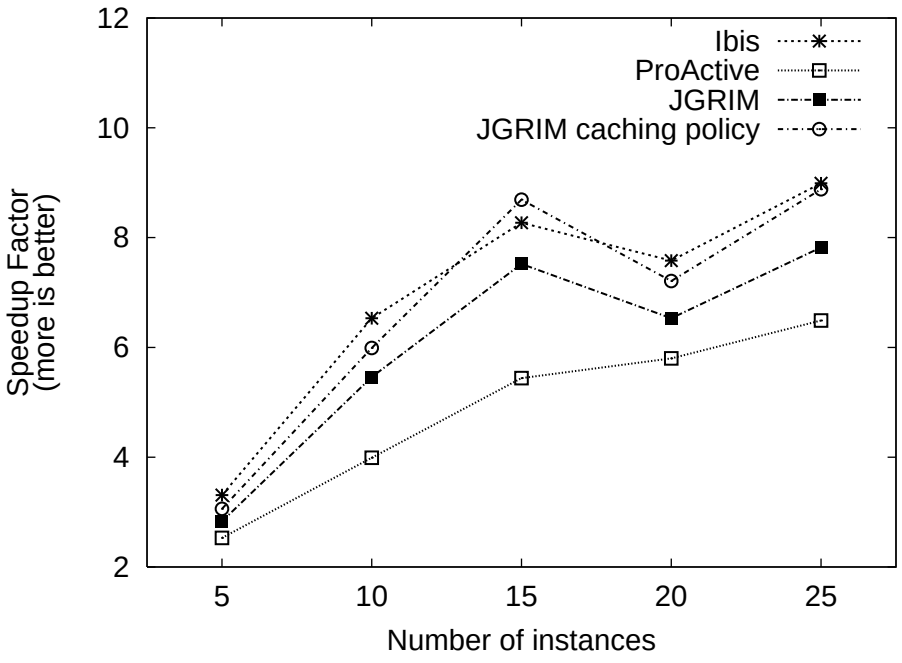


Figure 7.9: Speedup introduced by the gridified versions of the k-NN application with respect to the original implementation

In parallel computing, speedup refers to how much a parallel algorithm is faster than its sequential counterpart. Speedup is defined as T_1/T_p, where T_1 is the execution time of the sequential algorithm, and T_p represents the execution time of the parallel version of the algorithm on p processors. Similarly, Figure 7.9 depicts the speedups associated to the k-NN application, that is,

the time necessary to execute the original implementation –which sequentially classifies instances based on a file-based dataset– over the time required to run the Grid-enabled k-NN implementations. All test batteries corresponding to the original application were run on "VPNServer", which is a fast machine.

Note that, despite achieving different speedup levels, the speedup "functions" of the Ibis and the two variants of JGRIM seemed to have the same behavior. Basically, this is because these implementations share the same execution model (i.e. the job stealing mechanism of Ibis) to parallely access the dataset service and classify individual instances, which is the stage of the whole classification process where implementations spent most of the time. On the other hand, ProActive appeared to gain efficiency as the number of instances increased, but clearly more experiments should be conducted to corroborate this trend.

Overall, the implications of the resulting speedup is twofold. First, the original application certainly benefited from being ported to the Grid. Second, and more important, JGRIM achieved speedups levels that are competitive to those achieved by Ibis and ProActive. Specifically, using the caching policy allowed the JGRIM application to achieve, together with the Ibis version, the highest levels of performance.

7.2.2.2 Comparison of network traffic and packets

Figure 7.10 illustrates the total network traffic (measured in GB) that was generated across the Grid when running a test battery, that is, ten runs of an individual gridified implementation of the k-NN application. This traffic computes the amount of data that was sent from any of the Grid machines to another machine residing in either the same or an external cluster. Although LAN communication is significantly cheaper than communication between machines that are connected through the Internet, the traffic associated to intra and extra cluster communication was not discriminated because the latter just represented a very small fraction of the total traffic. In other words, even when the Grid-aware applications are in essence service-bound, services are always accessed through local links.

Specifically, the traffic destinated to extra cluster communication was only 23.46 MB (Ibis), 12.98 MB (ProActive), 30.06 MB (JGRIM, without caching), and 26.95 MB (JGRIM, with caching). The relation among these values clearly cannot be generalised, but they are certainly a consequence of the way each toolkit manages the execution of applications at the platform level. In Ibis, idle machines –that is, machines not executing a spawned computation– periodically generate requests to other nodes of the Grid participating in the execution of the application to get an unfinished computation from those nodes. In this way, requests originated at any machine can have a remote node as its target. JGRIM applications inherit this behavior since parallelization is currently based on the Ibis execution services. Therefore, extra cluster traffic in JGRIM comprises the data in remote job stealing requests, plus the traffic related to service brokering and diffusion of host profiling information. Lastly, the ProActive implementation generated less extra cluster communication, but its code resulted in a mix of application logic

and many instructions to manually start the classification of individual instances on specific hosts.

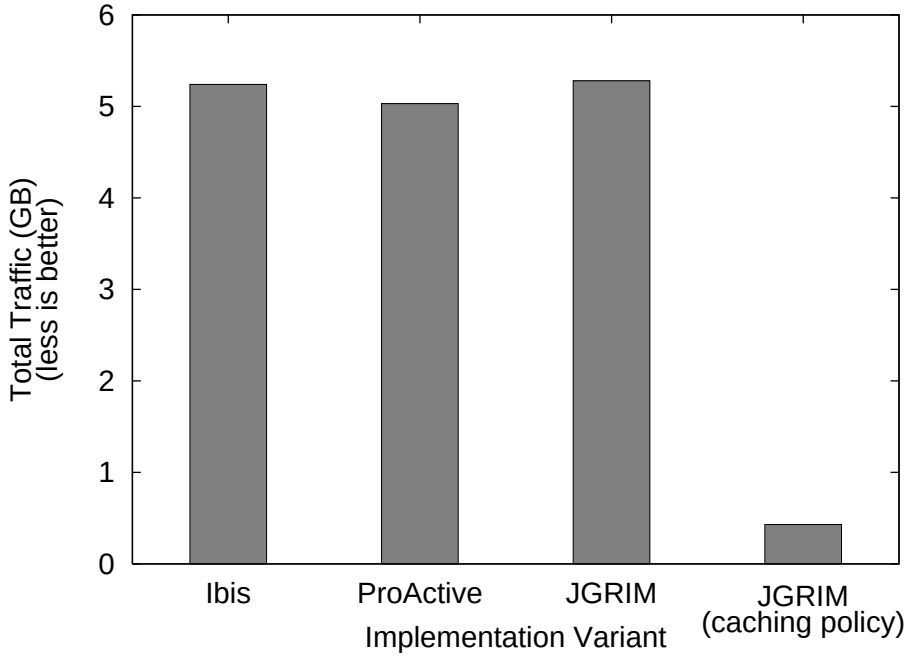


Figure 7.10: Total traffic (GB) generated by each variant of the k-NN application

All in all, the plain JGRIM implementation only added an overhead of less than 1% to the total traffic generated by the Ibis implementation, which represented about 4 MB of extra traffic per test. Note that the overhead is totally acceptable because part of this extra traffic was destinated to support Grid service brokering, whose benefits have been already discussed. Furthermore, the generation of the traffic related to the diffusion of profiling information gradually spread over the entire test time, which, unlike bursty communication, helps in avoiding network congestion. In short, these results show that the JGRIM solution did not incur in high communication overheads with respect to the application using the underlying, wrappered Ibis execution services.

The JGRIM implementation with the caching policy not only showed very competitive performance levels with regard of the rest of the implementations, but also allowed the application to drastically decrease the network traffic. Particularly, it reduced the traffic generated by the other versions (including the plain JGRIM implementation) by around a 92%. Obviously, these gains are a direct consequence of reducing the number of accesses to the dataset service. Nevertheless, as explained previously, the most interesting aspect of the caching mechanism is that it was implemented without altering neither the structure nor the logic of the original code.

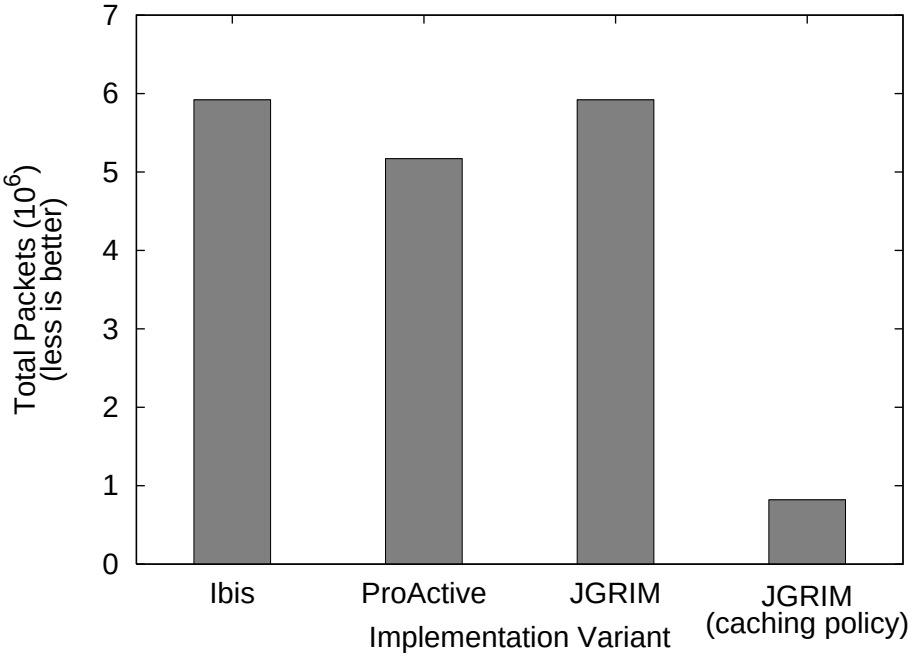


Figure 7.11: Amount of TCP packets transmitted by Grid machines during each test battery

Finally, Figure 7.11 illustrates the total network packets generated by each variant of the k-NN application during an entire test battery. The amount of packets generated by the Ibis and the plain JGRIM variants were very similar. As expected, the JGRIM version using the policy for dataset caching reduced the amount of packets generated by the ProActive solution by nearly a 84%.

7.3 Panoramic Image Restoration

Panoramic imaging is a subarea of photography that aims at creating images with very wide fields of view. Specifically, panoramic imaging is thought to capture a field of view of at least that of the human eye, while maintaining detail and consistency across the entire *panorama*. A panoramic image is thus a complete view of an area that is composed of smaller images.

This section describes the gridification of an application for restoring panoramic images located at a remote image repository. Algorithmically, the application works by downloading a specific image from a known host, splitting it into one or more smaller images, and finally restoring each part of the original image by applying a CPU-intensive restoration filter[9] that implements the algorithm proposed in (Tschumperlé and Deriche, 2003).

[9]Both a C++ library and binary executables implementing this filter are available at http://cimg.sourceforge.net

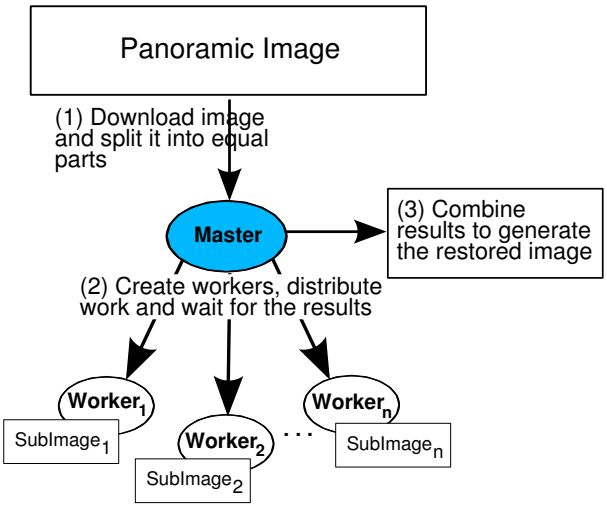


Figure 7.12: Overview of the panoramic image restoration process

As illustrated in Figure 7.12, the non-gridified version of the application was implemented according to the well-known master-worker pattern. Restoration of individual parts of an image are handled by different threads (i.e. workers), thus taking advantage of multi-core machines. The master is responsible for downloading the image, creating the workers, and assigning to each one of them an individual part of this image to filter. After all parts have been processed, the master combines the results into one single restored image. The idea is then to gridify this application so that workers can solve its assigned task by concurrently executing on the nodes of the Grid.

Figure 7.13 illustrates the class diagram corresponding to the original implementation of the restoration application. The `ImageRestorer` class implements the master component. Workers keep a reference to the master for signaling purposes upon completion of their assigned work. As we will explain later, this association represented one of the most problematic obstacles towards gridifying the application with the middlewares. Image splitting/merging behavior was implemented by the `RestoreUtils` class, whose implementation code is based on the functionality provided by the Java Advanced Imaging (JAI) and Java Image IO media APIs from Sun[10].

The image repository (FTP server) was installed on the "VPNServer" machine located at the ISISTAN cluster. To experiment with the application, the image repository contained an RGB image in JPEG format of about 2.4 MB (10120 x 2200), and four more pictures obtained from rescaling this image down to 76, 63, 37 and 18 percent. Rescaled images resulted in a size of 1.8 MB (9100 x 1980), 1.5 MB (8080 x 1760), 900 KB (6060 x 1320) and 400 KB (4040 x 880), respectively. In the original implementation of the application (and therefore also in their gridified counterparts), images were split by the master into twenty equal parts. It is worth

[10]http://java.sun.com/javase/technologies/desktop/media

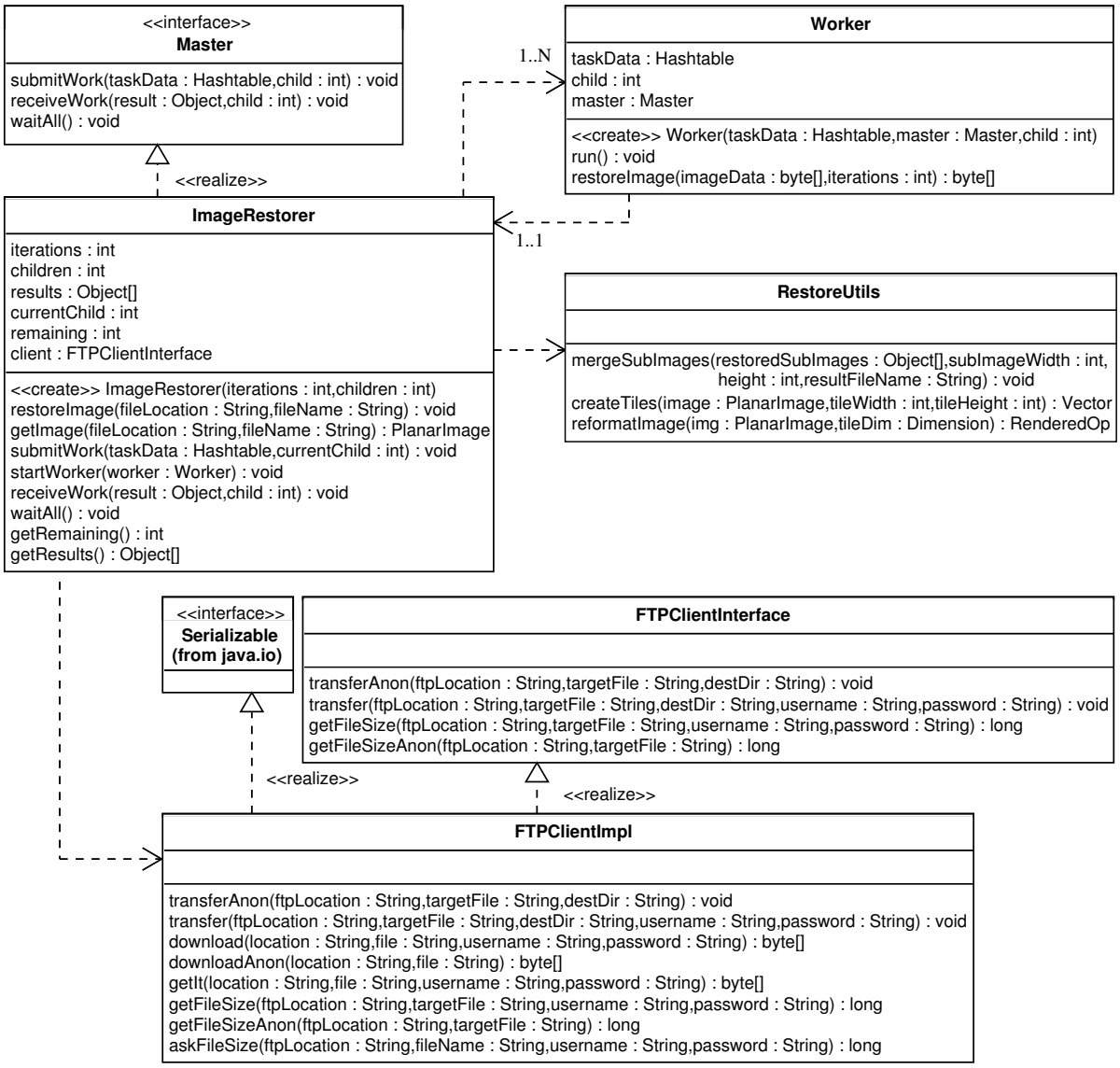


Figure 7.13: Class diagram of the restoration application

pointing out that, although these images are relatively small in size, the restoration algorithm is extremely CPU-intensive, thus they effectively served to enable the algorithm to produce long-running computations.

Sections 7.3.1 discusses the gridification of the restoration application by using Ibis, ProActive and JGRIM. Then, Section 7.3.2 presents experiments evaluating aspects related to performance and bandwidth consumption.

7.3.1 Gridification Effort

In order to evaluate the characteristics of both the original and the gridified versions of the application, the code metrics explained in Section 7.2.1 were used. Table 7.5 shows the resulting metrics for the Ibis, ProActive and JGRIM implementations of the application, and a variant

of JGRIM using a policy to move the application to the image repository location and access images locally, thus avoiding the downloading process.

As illustrated in the table, the size of the applications did not significantly vary among each other. The ProActive version introduced a code overhead of 24%, while the Ibis and JGRIM implementations reduced the original code by about 6%. Unfortunately, the negative overhead values of Ibis and JGRIM resulted from removing several source code lines directly related to the logic of the original application, specifically, code implementing synchronization and coordination behavior between the master and its workers. Nevertheless, the ProActive application preserved the interaction scheme originally designed to manage communication and coordination between master and worker components. In fact, one of the first official uses of ProActive was precisely for developing master-worker Grid applications. On the other hand, the JGRIM solution was also able to provide a fitting alternative to coordinate master and workers.

k-NN version	TLOC	PLOC	NOC	NOI	NOT	Code overhead (%)
Original	241	–	3	1	37	–
Ibis	227	5	**3**	1	40	-5.81
ProActive	299	17	4	1	46	24
JGRIM	**226**	**0**	**3**	1	**36**	**-6.22**
JGRIM (move)	233	1	4	1	**36**	-3.32

Table 7.5: Gridification of the restoration application: code metrics

Master and workers in ProActive were implemented as active objects. ProActive provides a mechanism so that references between active objects are transparently managed by means of specialized proxies that hide the physical location of active objects. When a method call is performed on an active object A, the invocation is transparently forwarded to the actual object instance representing A, regardless where it is currently located. Consequently, worker instances in the ProActive application were straightforwardly implemented to send their results to the master object, to which a reference is obtained when a worker is first created. However, while ProActive greatly preserved the original interaction structure, some application and API code to implement active object deployment/task management had to be supplied, which in turn affected PLOC.

In contrast, unless explicitly managed by means of its communication API, Ibis does not automatically maintain references between distributed application objects. In consequence, spawning a computation W from within a method of a master M which passes itself as an argument to W results in losing the reference to M in W. Another alternative that was explored, similar to the k-NN Ibis implementation, involved the use of a shared object to indirectly communicate the master and the workers. Unlike the k-NN application, task results coming from workers are

processed subimages that may be pretty large in size. Since Ibis automatically broadcast an individual write to a shared object to its distributed copies, employing this mechanism would clearly require too much network resources. This can be avoided by maintaining a list of shared objects, each handling the communication of the master and exactly one worker. However, this would had made the implementation of the application more difficult.

To overcome the above situation, the iterative-like work submitting structure of the original application was transformed to a recursive algorithm, in which each leaf of the execution three spawns a computation that is in charge of processing an individual portion of the image being restored. The code structure was basically the one presented in Section 7.2.1.3 (JGRIM implementation of the k-NN algorithm). A code snippet of the master component is presented next:

```
void submitWork(Vector tiles){
  ...
  Vector results = submitWork(tiles, 0);
  // Ibis synchronization primitive
  super.sync();
  ...
}

Vector submitWork(Vector tiles, int currentChild){
  if (tiles.size() == 1){
    byte[] subImagePixels = (byte[]) tiles.firstElement();
    Hashtable data = new Hashtable();
    data.put("pixels", subImagePixels);
    data.put("iterations", iterations);
    Worker worker = new Worker(data, currentChild);
    Vector result = new Vector();
    result.addElement(worker.run());
    return result;
  }
  int mid = tiles.size()/2;
  Vector tilesStart = grabElements(tiles, 0, mid);
  Vector tilesEnd = grabElements(tiles, mid, tiles.size());
  Vector res1 = submitWork(tilesStart, currentChild);
  Vector res2 = submitWork(tilesEnd, currentChild + mid);
  // Ibis synchronization primitive
  super.sync();
  res1.addAll(res2);
  return res1;
}
```

The JGRIM implementation demanded some modifications that can be grouped in two categories. On one hand, similar to its k-NN counterpart, a recursive `submitWork` method was implemented to take advantage, through a self-dependency, of the execution and parallelization services provided by the Ibis platform. In this case, the transformation is absolutely necessary since JGRIM discourages explicit referencing between application components (i.e. master and workers), because in practice it leads to poor reusability and portability of both components and even applications across the Grid.

On the other hand, a dependency named *client* from the master component (implemented by the `ImageRestorer` class) to the FTP client component was declared. In consequence, direct access to this component were replaced across the source code of `ImageRestorer` by calls to the corresponding `getclient` method. As in the case of the k-NN application, other data reader or "downloader" component can be used for the application (e.g. GridFTP instead of plain old FTP), as long as this new component adheres to the service interface required by the master component, represented by the `FTPClientInterface` Java interface.

A place of the application where mobility can be useful is when accessing the remote image repository. Generally speaking, application mobility can bring significant benefits in terms of decreased latency and bandwidth usage when applications are moved to locally interact with remote data. In this sense, a variant of the JGRIM application was built by simply configuring the following policy to the *client* dependency:

```
public class AlwaysMovePolicy extends PolicyAdapter{
  public void executeBefore(){
    // Force moving the agent to the repository location
    getOwnerAgent().moveTo("VPNServer");
  }
}
```

Lines one and two, although pointing to an object type and a method defined in the JGRIM API respectively, is not considered by PLOC, since the skeleton for any policy is automatically generated at gridification time. Conversely, line four has been manually added to instruct the JGRIM runtime system to migrate the MGS upon access to methods declared by the FTP component. The `getOwnerAgent` method is provided by the policy framework to get a reference to the application object that represents the executing MGS. That is to say, when the application requests to download an image file, the code within the `executeBefore` method is executed, thus the application state is transparently transferred to the site hosting the data. It is worth noting that a call to $moveTo(S)$ has no effect at all if the requesting application is already located at S. Finally, more complex policies could be implemented to further increase efficiency. For example, it might be interesting to move the agent only in those cases where the size of the target image file exceeds a certain threshold.

In contrast, mobility for the Ibis version was not possible, since Ibis implicitly manages migration of spawned computations between machines and does not let applications to explicitly control

this feature. On the other hand, ProActive do provide a `migrateTo` API method. However, this primitive implements a weak migration mechanism for active objects, thus placing a burden on the developer since inherently complex and lengthy code to manually maintain the execution state of the application must be supplied. Due to this fact, it was decided not to add mobility to the ProActive application so as to keep the TLOC metric low, and to simplify application programming.

Table 7.6 shows the bytecode information associated to the different versions of the restoration application, and the anatomy of the test code used to launch the applications. Since the restoration application required less Grid functionality than the k-NN application (i.e. parallelization only), bytecode information can be used to get a more accurate estimation of the minimum bytecode overhead introduced by each middleware for the same application. Since ProActive instruments applications at runtime, bytecode overhead after compiling the gridified application was less than the one introduced by Ibis and JGRIM. Furthermore, the JGRIM implementation introduced a bytecode overhead of 15% to the Ibis solution. This is an undesirable nevertheless acceptable overhead given the added value of JGRIM applications in terms of component reusability and decoupling, and the possibility of leveraging the services provided by other Grid middlewares. In any case, dynamic class instrumentation could be employed to cut down some of this overhead.

k-NN version	# .class	Bytecode size	Bytecode overhead (%)	Test lines/PLOC
Original	4	12731	–	–
Ibis	10	30515	139	32/0
ProActive	5	16852	32	48/11
JGRIM	11	35137	175	34/7

Table 7.6: Characteristics of the restoration applications upon execution on the Grid setting

Figure 7.14 shows the resulting *GE* for the different versions of the image restoration application. To make these results comparable to the ones described in Section 7.2.1.4, *GE* values were scaled down by using the same factor (10^3). According to the *GE* indicator, the restoration application was easier to gridify than the k-NN application. This situation truly makes sense because after gridification the former utilizes less Grid functionality, does not use functional Grid services, and includes less source code to perform application tuning.

The resulting *GE* was 0.046 (Ibis), 0.095 (ProActive) and 0.042 (JGRIM), which indicates that gridifying with ProActive demanded more effort than doing so with Ibis or JGRIM. Specifically, JGRIM reduced *GE* of ProActive by about 55%, whereas Ibis decreased this effort by 51%. As illustrated in the figure, most of the effort when gridifying with Ibis and JGRIM was invested in providing code to launch the execution of the Grid-aware application. In any case, the plugging effort can be drastically reduced by providing better tools to easily run the Grid--enabled applications. In fact, one important feature that is currently missing in JGRIM and will be supplied in the near future is concerned with the provision of desktop as well as Web

interfaces to graphically launch and monitor the execution of gridified applications.

As observed, the reimplementation effort in JGRIM was slightly greater than in Ibis. However, one important point must be clarified. The restoration application is inherently a pure parallel application, and is not intended to take advantage of other Grid resources and services. These characteristics makes the application ideal to be implemented by using the parallelization services of Ibis. Furthermore, JGRIM materializes an approach whose goal is precisely to isolate applications from the way Grid services are accessed, and therefore how parallelization is performed, which translates in a little extra effort at gridification time. Nevertheless, this small overhead is completely acceptable given the benefits of JGRIM in terms of maintainability and portability of a gridified application. In the experiments, the logic of the JGRIM restoration application was absolutely clean of Grid API code and decoupled from the specific execution service, thus workers can be configured to be run by means of other execution services (e.g. threads, Globus jobs, ProActive active objects, etc.) with little effort and without source code modification.

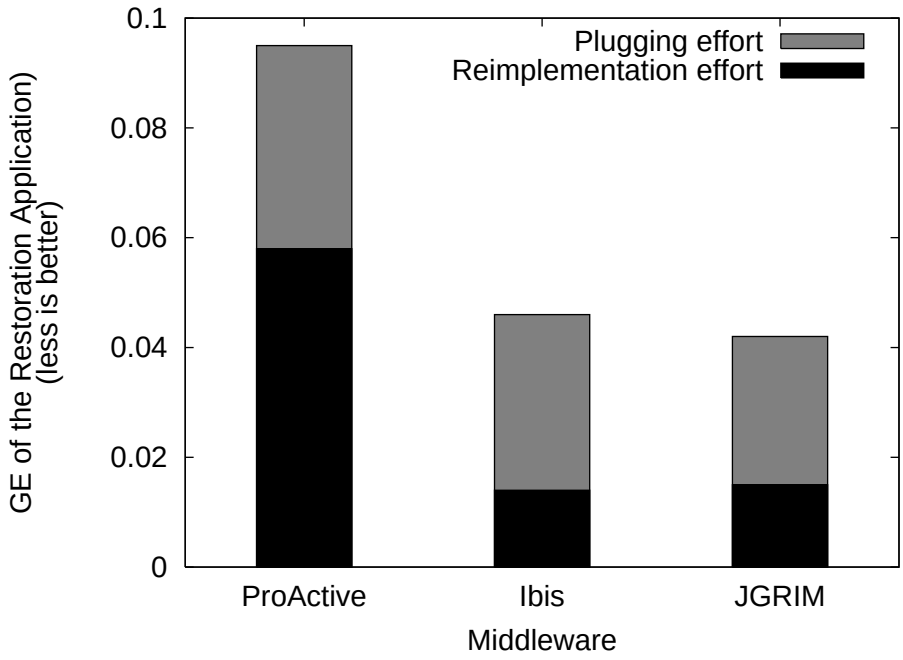


Figure 7.14: Gridification effort for the restoration application

Since both the k-NN and the restoration application were gridified by the same person, and this former was ported first, it is clear that the developer had knowledge about each Grid API at the time the gridification of the latter started. In this light, GE was computed by considering $APIFactor = 0$. The goal of this adjustment is to model the fact that programmers often face a steep learning curve when employing new Grid technologies, but as they use them, the learning effort tend to disappear. Learning curves corresponding to the three middlewares used

in the experiments were assumed to be similar to each other because of the developer's strong background on distributed and parallel programming, and the similarity/correspondence in the API complexity/target programming language.

7.3.2 Performance Analysis

This section reports the performance tests that were conducted based on the various versions of the restoration application presented previously. To better understand the runtime behavior of these applications, and set the basis for a more detailed comparison, a simple benchmark based on the original application was prepared. To this end, the aforementioned image repository was used.

The purpose of the benchmark was to have an estimation of the time (on average) necessary to download and to restore an image from the repository. These two factors were estimated separately. On one hand, the average transfer time (ATT) was measured by performing five different downloads of each image (located at the "VPNServer" machine of the ISISTAN cluster) from the remote "ale" machine residing at the Ale cluster. To avoid unnecessary noise, the tests were run under low network load. Table 7.7 shows the resulting transfer times. From the table, it can be seen that, keeping in mind that all transfers were carried out through a public WAN link, experienced deviations were in the range ($0.4 - 2.0$ seconds), therefore the network benchmark has an acceptable confidence level.

Image (KB)	Size (KB)	ATT (secs)	Std. Dev. (secs)	Avg. transfer rate (KB/s)
Image1	435.40	17.03	0.43	25.56
Image2	903.12	34.56	1.70	26.13
Image3	1515.56	55.71	0.51	27.20
Image4	1819.16	66.98	0.53	27.16
Image5	2409.18	94.21	1.98	25.57

Table 7.7: Restoration application: network benchmark

For practical reasons, the time required to enhance an image was estimated by running the restoration process on a random image portion (i.e. an horizontal component of the panorama) of a size of 1/20 of the original image, and multiplying the elapsed time by 20. Note that this is exactly the same criteria for splitting an input panoramic image that is employed by the gridified versions of the application.

Table 7.8 shows the total estimated execution time (in minutes), calculated as twenty times the average execution time (AET) among five runs of the restoration process on a portion of an image. All tests were run on "VPNServer". Again, little standard deviation –specifically in the range of $0.55 - 1.35$ seconds– was obtained, thus the estimations will serve to our purposes.

The estimated total execution time (TET) required to download and process a complete image is shown in Table 7.9. On average, the lower and upper bounds of the amount of KBytes processed

Image (1/20)	AET (min)	AET * 20 (min)	Std. Dev. (secs)
Image1	0.44	8.80	0.56
Image2	0.99	19.97	0.67
Image3	1.78	35.63	0.68
Image4	2.34	46.80	0.99
Image5	2.79	55.80	1.32

Table 7.8: Restoration application: performance benchmark

per unit time (minute) was in the range of about 38 and 48, respectively. The table also shows the time increment ratio between any image and its immediate preceding image.

Image	TET (min)	$\mathbf{TET}_n/\mathbf{TET}_{n-1}$	Throughput (KB/min)
Image1	9.08	–	47.95
Image2	20.55	2.26	43.95
Image3	36.56	1.78	41.46
Image4	47.92	1.31	37.96
Image5	57.37	1.20	41.99

Table 7.9: Total execution times and throughput of the original restoration application

The experiments using the gridified versions of the above application were performed by processing each image ten times, and then computing the resulting TET (in minutes) and throughput (in KB per minute) accordingly. As in the case of the k-NN application, network resources consumed by applications, namely the generated TCP traffic (in MB) and the amount of data packets, were measured by using the tcpdump and Wireshark tools. In addition, all tests were initiated in the "ale" machine.

7.3.2.1 Comparison of TET

The TET values obtained from the different versions of the restoration application are depicted in Figure 7.15. Specifically, the figure illustrates the behavior of the Ibis, ProActive and JGRIM implementations, and the variant of JGRIM employing the policy that moves the application to the image repository location when an individual image is downloaded. As explained, the implementation of the policy was almost effortless, because only involved the use of a single line of code for moving the MGS to "VPNServer". The graphic includes the errorbars in the y axis corresponding to the standard deviation of the elapsed times.

From the figure, it can be observed that JGRIM performed very well and experienced performance levels similar to its related approaches. Specifically, the function that defines its associated

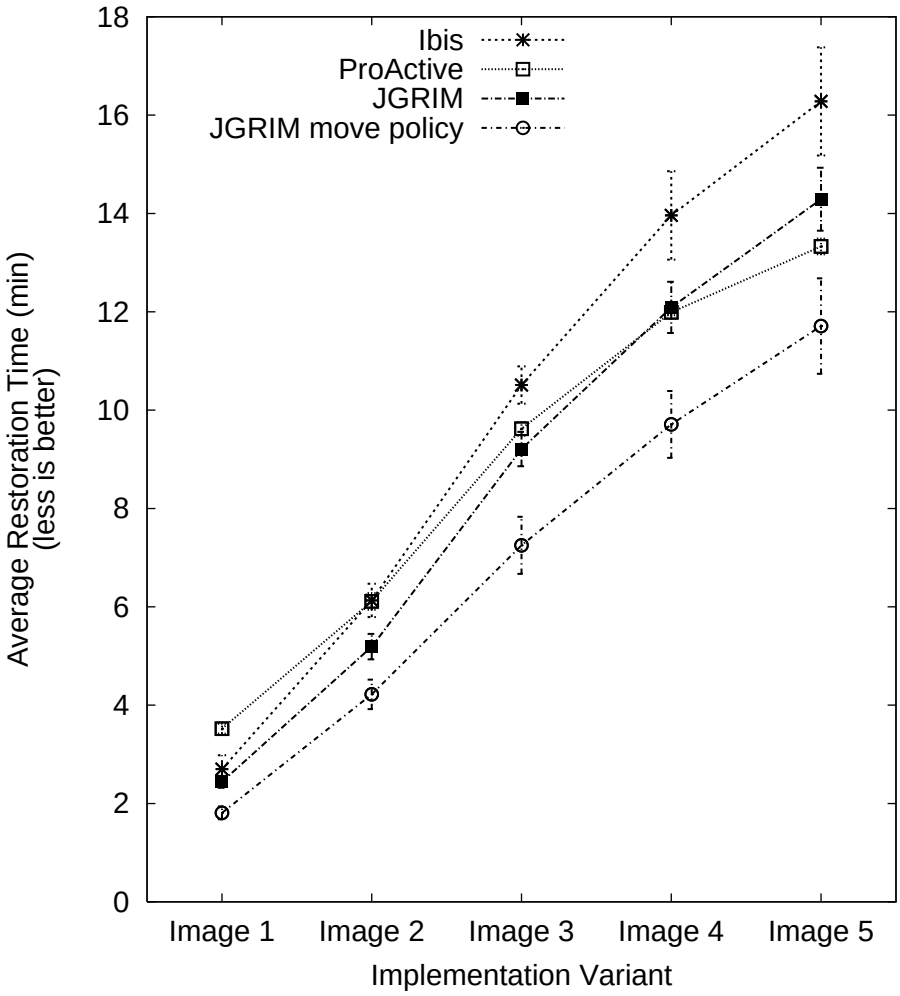


Figure 7.15: TET (min) of the restoration application

TET appears to be close to the one describing the TET of the ProActive version. In addition, the variant of JGRIM using the policy for moving the MGS brought significant benefits in terms of increased execution performance. Another interesting fact is that coding this policy was very easy. Also, as policies are individual software components that are non-intrusively injected by JGRIM into applications, they do not affect the code of the application logic and can be reused in other applications.

A result that may confuse the reader is that the JGRIM version of the application performed better than the Ibis variant, even when the former used the Ibis services as the underlying support for application parallelization. The reason of this is that the code that is interpreted by the Ibis runtime in either cases is subject to different execution conditions. On one hand, the implementation of the Ibis version comprises the application logic *plus* the code (main method) to run the application, which is called by the Ibis runtime to carry out the handshaking process among Ibis hosts to actually start and cooperatively execute the application. On the other hand, upon the execution of a self-dependency method, an Ibis object is created and sent by the JGRIM platform to an already deployed Ibis network, which is running a pure Ibis application (i.e. Ibis server) that is able to execute other Ibis objects.

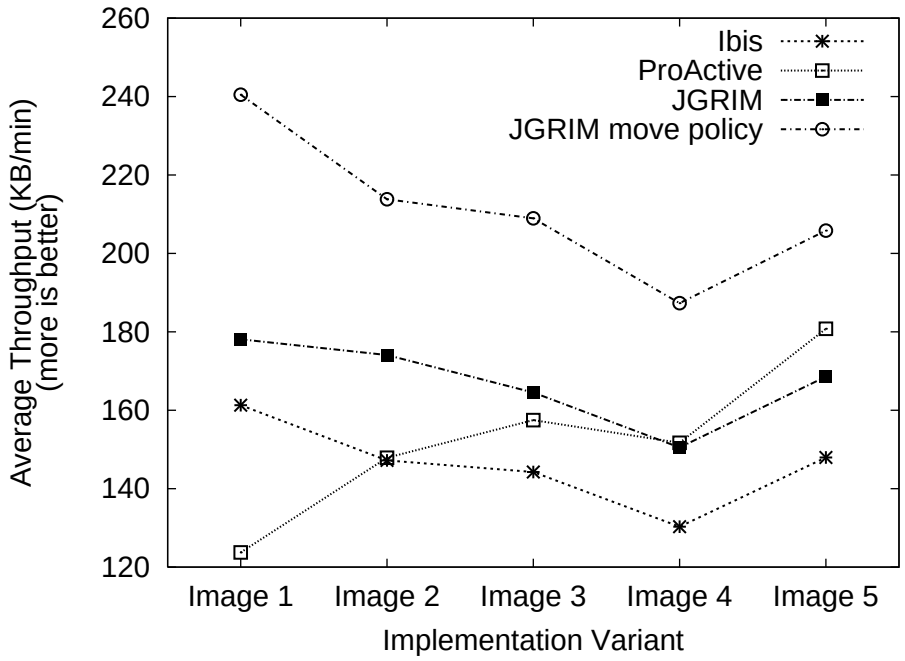


Figure 7.16: Throughput (KB/min) of the restoration application

Figure 7.16 shows the throughput achieved by the different variants of the application, calculated as the average amount of data processed per time unit, that is, the amount of image data (KB)

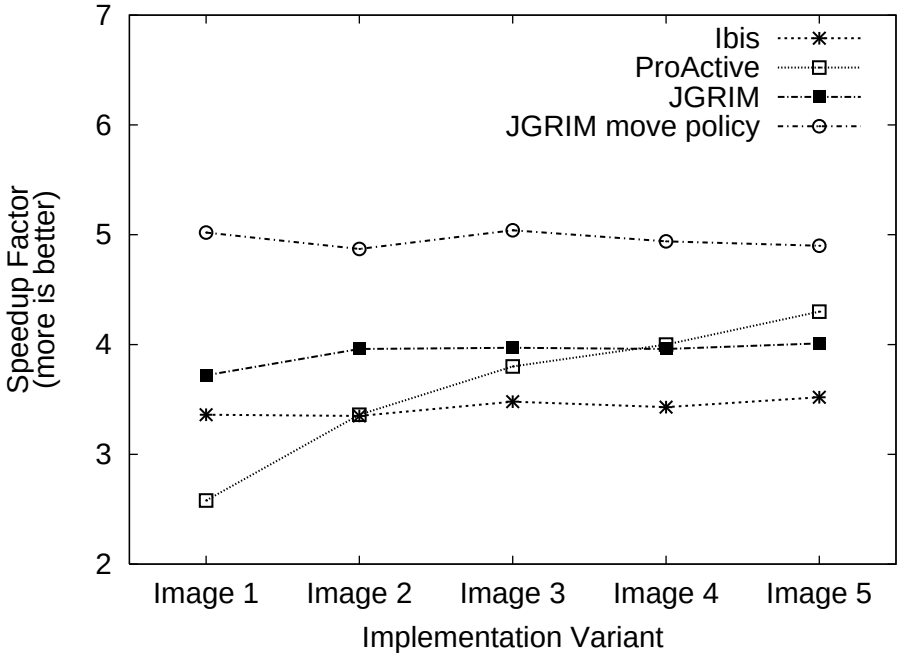


Figure 7.17: Speedup introduced by the gridified versions of the restoration application with respect to the original implementation

restored per minute. As reported, the JGRIM application offered a throughput level similar to its competitors, and even better levels were achieved by using policies.

In addition, Figure 7.17 depicts the speedups associated to the restoration application, calculated as the time necessary to execute the original implementation (estimated by the above benchmark) over the time required to run the Grid-aware versions. Remarkably, employing a very simple policy allowed the JGRIM application to outperform the Ibis as well as the ProActive implementations.

7.3.2.2 Comparison of network traffic and packets

To profile the network resources utilized by the variants of the restoration application, the network traffic (measured in MB) and TCP packets generated during the test runs were measured. In this sense, Figure 7.18 shows the total traffic in MB generated by each variant during the whole experiment, that is, the traffic accumulated throughout the ten runs that were performed to compute the average execution time. Moreover, traffic has been discriminated into two types: *intra cluster*, which counts the LAN traffic generated to communicate any pair of machines residing in the same cluster, and *extra cluster*, which measures the total network traffic for sending data between two machines that belong to different clusters. Since clusters are connected to each other by Internet links, it is extremely desirable to generate little extra cluster traffic because Internet communication is several orders of magnitude more expensive than local communication.

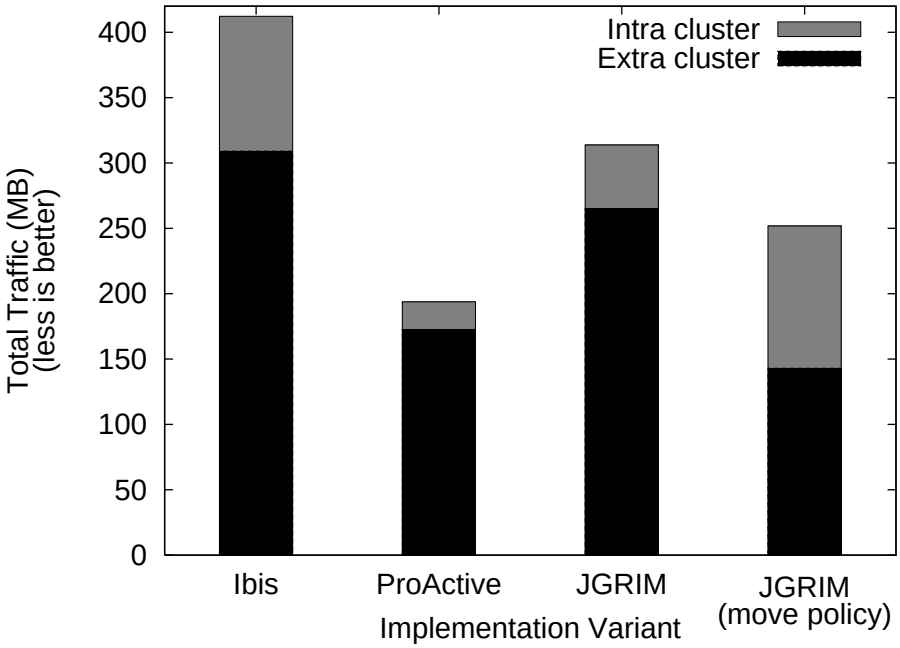


Figure 7.18: Total traffic (MB) generated during the entire experiment

The plain JGRIM variant of the application generated less and more traffic than the Ibis and ProActive implementations, respectively. However, extra cluster communication in the JGRIM version represented the 84% of its total traffic, against a higher value of 89% for ProActive. Furthermore, using policies allowed JGRIM to further reduce the total traffic. Even more interesting, the percentage of extra cluster communication with respect to the total traffic in the policy-based JGRIM application dropped down to 57%. In addition, this latter reduced the extra cluster communication of ProActive by almost 18% (30 MB). These facts evidence an important aspect of JGRIM: policies are certainly useful not only to achieve higher performance but also to make a better use of Grid network resources.

There are other interesting points that can be observed from the figure. First, the middleware that generated the least amount of total network traffic was ProActive. Unlike the rest of the implementations in which nodes randomly try to steal jobs from their peers and thus many messages may be sent before actually get a job to execute, ProActive promotes the use of a job submission model in which the application decides which node (i.e. which active object) should handle the execution of the next unfinished job. In consequence, less traffic is generated, but a significant amount of code have to be provided by the programmer in order to support this mechanism. Second, similar to the case of TET discussed in the previous subsection, JGRIM incidentally used less network resources than Ibis since both implementations access the services provided by the Ibis platform according to a different application execution scheme.

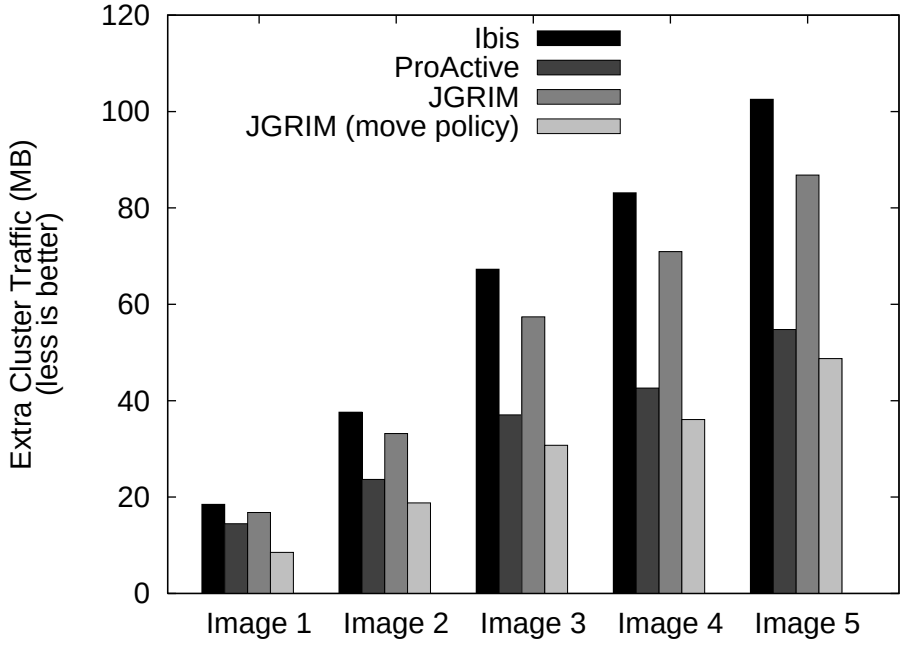


Figure 7.19: Total traffic (MB) generated during the entire experiment

As a complement, Figure 7.19 illustrates the extra cluster network traffic (also in MB) generated by each application during the restoration process of an individual image. Naturally, the total traffic results in higher values as the size of the input image increases. In all cases, the variant of JGRIM with the move policy was the most efficient application in terms of network usage. Lastly, data tables of the total packets destinated to intra and extra cluster communication across Grid nodes when restoring individual images can be found in Appendix D. The percentage of extra cluster packets with respect to the total generated packets was 52 (Ibis), 89 (ProActive), 72 (JGRIM) and 57 (JGRIM with policy). In consequence, the non-policy and policy-based JGRIM implementations of the application experienced network performance levels –with respect to TCP packets– similar to that of ProActive and Ibis, respectively. The amount of packets used to carry out local communication can be ignored, as they are subject to very small latency values compared to extra cluster communication. Finally, in both JGRIM implementations, a high percentage of the extra cluster packets corresponded to communication performed by the Ibis execution service.

7.4 Conclusions

This chapter described in detail some experiments that were performed with JGRIM in order to validate GRATIS. Specifically, the validation was carried out by comparing JGRIM with Ibis and ProActive, two Java-based platforms for Grid application development and execution that materialize alternative approaches to gridification. Experiments show that gridifying applications with JGRIM demands less effort from the developer, and effectively preserves the application

logic, thus producing Grid-aware code that is significantly easier to maintain, test, and port to various Grid settings and platforms. In addition, experiments suggest that, even when JGRIM may cause gridified applications to consume more Grid resources than their Ibis and ProActive counterparts, this do not directly translate into an irremediable problem, since policies can be used to easily and non-intrusively improve resource usage and to allow JGRIM applications to perform in a very competitive way with regard to related approaches.

Chapter 8

Conclusions and future work

The Grid is a globally distributed computing environment that virtualizes resources underneath an abstract layer providing special services through which applications can greatly improve their capabilities. The notion of "Grid" find its roots in an analogy with the electrical power grid infrastructure, since developers will take advantage of Grid resources by simply plugging conventional applications to the Grid. However, zero-effort portability of ordinary applications is still far from being a reality.

In this light, the previous chapters described a new approach for gridifying existing applications called GRATIS, whose goal is to solve the problems inherent to gridification not yet fully addressed in the literature. Essentially, GRATIS promotes separation of concerns between application logic and Grid behavior, combining the notions of Dependency Injection, service--oriented software and mobile services to facilitate the construction of API-agnostic, maintainable, portable and efficient Grid applications. Based on the concepts introduced by GRATIS, a middleware for gridifying applications called JGRIM was implemented. JGRIM allows the developer to easily gridify an application by mapping it to an MGS, which are basically mobile entities that live in the Grid. JGRIM injects MGSs with one or more metaservices to transparently interact with Grid resources, infrastructure services and other MGSs.

In order to experimentally validate the GRATIS approach, some applications were developed and then gridified by using JGRIM as well as other Grid toolkits designed for gridifying applications. Then, a detailed analysis of the incurred gridification effort in each case was carried out. In addition, a comparison of the different implementations in terms of their runtime behavior and resource usage was performed. To this end, a concrete Grid setting composed of several Internet-connected computer clusters was configured, which served as a testbed for running the applications. From the experiments, it can be seen that JGRIM, besides reducing the gridification effort and producing Grid-free application code, allows Grid-enabled applications to achieve levels of efficiency very similar or even better compared to applications gridified with the other middlewares.

The next section details the main contributions of the present work to the area of Grid Com-

puting. Then, Section 8.2 enumerates its limitations. Later, Section 8.3 describes prospective future research directions. Finally, Section 8.4 presents concluding remarks.

8.1 Contributions

This thesis introduces several contributions to the area of Grid Computing:

- It defines a novel, two-step approach to gridification that combines the benefits of the component-based and SOA programming paradigms in terms of modularity, reusability, loose coupling and interoperability, with the advantages of the Dependency Injection concept, which allows application components and distributed services to be transparently assembled together at the middleware level. GRATIS builds upon these notions to allow conventional applications to use Grid services with a minimum development effort. In addition, GRATIS prescribes an effective tuning support for gridified applications that is based on the notions of mobile software and *policies*, that is, a mechanism to non--intrusively and dynamically adapt the execution of a Grid application according to the characteristics of both the particular Grid setting being used and the specific Grid services accessed by the application.

- From a software engineering perspective, GRATIS has a positive impact on Grid application development as a whole, since its gridification process produces applications that possess several quality attributes that are highly desirable for any type of software system. Specifically, as the gridified code combines but do not *mix* the original application logic with the Grid functionality, testing and improving this logic outside a Grid setting becomes more convenient. As a corollary, since the gridified application logic is free from Grid-related code, it is easier to port the same code to various Grid settings and execution environments. In addition, there are also some gains in terms of legibility, since the gridified code is more clear and is completely unaware of the Grid, thus developers do not need to know in advance specific Grid technologies or APIs before making further improvements or extensions to the application code.

- Through JGRIM, which is a middleware that materializes the GRATIS approach, the thesis experimentally showed that it is possible to achieve –with an acceptable programming effort– good levels of response time and network resource usage for a gridified application without forcing the developer to explicitly alter the application logic. In other words, providing implicit interaction between application logic and Grid services and still let applications to execute efficiently is viable from a practical point of view. In addition, since concerns related to service interaction are completely handled at the middleware level, JGRIM may also serve as a toolkit to greatly simplify the implementation of service-oriented distributed applications in other domains, such as electronic commerce and e--business applications.

- The thesis specifies a taxonomic framework that attempt to help the Grid research community to better understand the different dimensions of the gridification problem. Particularly, the framework is composed of four taxonomies that characterize existing approaches to gridification in terms of application redesign, compilation unit reimplementation, gridification granularity, and the kind of Grid resources applications are able to transparently leverage after gridification. The framework proved to be very useful to set a common basis for analysis and comparison among the existing approaches aimed at addressing the problem of gridifying ordinary applications.

- GRIM is a generic model based on SOA concepts and separation of concerns that guides and facilitates the construction of mobile software and supporting platforms. Basically, GRIM models common interaction patterns between applications and resources in distributed environments, and prescribes mechanisms to easily make these interactions more efficient. The model has been materialized by JGRIM as well as by other middlewares for Internet development. In this sense, its versatility may help to facilitate its widespread adoption not only to build Grid and Internet applications but also to develop other kind of distributed systems, such as those running in local networks, computer clusters and supercomputers.

8.2 Limitations

A limitation of the GRATIS approach arises as a consequence of the assumptions made when gridifying applications. Specifically, GRATIS assumes input applications as being constructed under a component-based paradigm, comprising a number of components exposing and requesting services through well-defined interfaces. Although the component-based programming paradigm is very popular, the assumption does not likely hold for any kind of applications. In the context of JGRIM, this may be the case of legacy Java code or applications having a bad structure in terms of object-oriented design. Fortunately, the problem of componentizing existing object-oriented applications has been already addressed in the literature (Lee et al., 2003; Kim and Chang, 2004). In this sense, a similar approach could be followed to provide JGRIM with an extra transformation step to ensure, prior to gridification, that input applications are in fact component-based applications.

Even though JGRIM is a tool that simplifies the gridification of existing applications and produces code that is free from Grid functionality, its current implementation lacks proper support for configuring and deploying ordinary applications to a specific Grid setting, and also for monitoring the execution of gridified applications. Consequently, development support tools are currently being developed to make JGRIM easy to adopt and use. Specifically, a prototype plug-in for the Eclipse SDK has been implemented. The middle-term goal of this tool is to help developers in gridifying their applications by graphically defining dependencies, creating/associating custom policies to these dependencies, and configure the necessary parameters to perform

the execution of the application on a Grid, such as the logical network where it will reside, authentication credentials, UDDI information, etc. It is expected that, in the future, the plug-in will also let users to debug and inspect the execution state of gridified applications.

Another weak point of the implementation of JGRIM is concerned with the management of self-dependencies. In its current state, JGRIM allows developers to declare exactly one self--dependency in the main application component, whose operations are by default executed by means of the Ibis services. On the contrary, note that having many self-dependencies would allow gridified applications to simultaneously benefit from a wide variety of Grid execution services. For example, operations having very strict or hard performance requirements might be associated to efficient and fault-tolerant Grid execution services, whereas operations that are suspected to cause problems or to behave abnormally might be handled by execution services that additionally include a convenient support for performing application debugging and error diagnosis.

8.3 Future work

In the course of this work, several directions for future research have been identified. In the following paragraphs, some of them are described.

8.3.1 Enhancement of the Parallelization Support

An interesting point that deserves further attention is concerned with the current implementation of the parallelization and policy support of JGRIM. On one hand, the method spawning scheme employed to handle the execution of self-dependency operations and wait for the corresponding results could be extended to transparently parallelize the invocation of operations declared by other types of dependencies (e.g. to Grid functional services). In this way, parallelly contacting for example two Grid services S_1 and S_2, together with the usage of policies that customize the way interaction with services is performed, may bring significant benefits in terms of performance thus allowing JGRIM to produce more efficient Grid applications. However, it will be necessary to carefully study how this new support affects not only the existing metaservices but also the execution model for MGSs. For example, it is necessary to determine how to handle a situation in which a policy attached to S_1 decides to move the MGS to another host while the result of a parallel invocation to S_2 is still not available.

Similarly, the policy support could be also enhanced to provide facilities for customizing parallelization, such as controlling at runtime the way dependency operations are called (synchronously vs. asynchronously) or under what conditions the execution of a certain self--dependency operation should be spawned. This will allow programmers to better adapt the same application code –with few or even no modifications– to the underlying Grid environment by using parallelization only in those cases where it helps in improving performance. For example, it may be useful to control the number of spawns generated by the application when

executing on an Ibis network according to conditions such as network size, processing capabilities of its nodes, latencies, and so on. An interesting recent work in this line is POP-C++ (Nguyena and Kuonen, 2007), a parallel programming language based on C++ that provides inter-object and intra-object parallelism according to various method invocation semantics.

8.3.2 Decentralized Mechanisms for Service Discovery

Another issue that should be explored in the near future is the use of more efficient mechanisms for service discovery across a Grid. Currently, service invocation and discovery metacomponents in JGRIM basically operate on a client-server basis: when an MGS requests a certain service, an appropriate service instance is retrieved by first contacting and then querying a specific service registry. However, tomorrow's Grids will offer thousands or perhaps millions of services to user applications. Additionally, as Grids evolve, the number of potential clients for these services will increase at a very high rate. In this context, achieving good scalability will be crucial, therefore rendering solutions for service discovery based on centralized schemes absolutely inappropriate. At present, approaches to decentralized service discovery include the use of UDDI node federations (*clouds*) and P2P technologies (Garofalakis et al., 2006), among others. Consequently, an important future line of research involves to incorporate in JGRIM a service discovery mechanism that follows either of these approaches. An incipient work in this line is the extension of the GMAC P2P protocol (Gotthelf et al., 2005) with service discovery capabilities.

8.3.3 Ease of Deployment

Although the analysis throughout this work has been explicitly centered around the notion of gridification as the process of transforming the source code of an application to run on the Grid, an aspect that deserves special attention is the amount of configuration that may be necessary to truly make this transformation happen. In a broader sense, gridifying an application is not only concerned with making conventional source code Grid-aware, but also with supplying some Grid-dependent configuration in order to run the adapted application, which usually ranges from application-specific parameters (e.g. expected execution time and memory usage) to deployment information (e.g. number of nodes to use). Sadly, this demands developers to know in advance many platform-related details before an application can take advantage of Grid services.

As gridification methods evolve, difficulties in gridifying ordinary applications seem to move from adapting source code to configuring and deploying Grid-aware applications (Mateos et al., 2007a). For example, this fact is evident in those approaches (e.g. GEMLCA, GRASG, XCAT) where code modification is not required but deployment becomes difficult. Nevertheless, the problem of simplifying the deployment of Grid applications has been acknowledged by the Grid research community. For instance, a ProActive application can be executed on several Internet-connected machines by configuring and launching the application at a single location. In this sense, future research in JGRIM will be also oriented towards make the launching, tuning and

deployment of gridified applications onto any Grid *easier*. One initial step towards reaching this end includes the Eclipse plug-in for gridifying applications mentioned above.

8.3.4 Language Independence

The approach to gridification described in this thesis is strongly based on notions from component-based software development, Dependency Injection and object-oriented programming. In addition, GRATIS has been at present materialized only for the Java language. In this sense, future research will explore the viability of materializing GRATIS concepts in other languages implementing alternative programming paradigms, such as compiled languages like C++ or interpreted languages like Python, Ruby or Perl. The main motivation of this task is that many of these languages (specially C++) are being extensively used for programming Grid and parallel applications. Note that materializing GRATIS in a different language will require to study whether the language supports the necessary core features such as application migration, function/method execution introspection, Web Service invocation capabilities, an so forth. Fortunately, many frameworks supporting the DI pattern for a variety of languages already exist, such as Autumn Framework and QtIocContainer (C++), Spring Python and PyContainer (Python), Copland and Needle (Ruby), and IOC Module (Perl). In addition, APIs for interacting with remote services as well as libraries implementing process migration techniques for some of these languages also exist. In this way, developers will be able to take advantage of the benefits of GRATIS for gridifying their applications and at the same time using programming languages of their choice.

8.3.5 Security

Finally, an important feature still missing in the solution described in this thesis is concerned with security provisioning. Since GRATIS applications can potentially travel across the boundaries of different administrative domains looking for Grid services and other applications, security is crucial to guarantee the integrity of both traveling applications and resource providers. The main factor that has hindered the widespread adoption of mobile agents is precisely their security problems (Kotz and Gray, 1999). Nevertheless, the problem of supporting security in mobile agent systems has been widely acknowledged and extensively investigated (Jansen, 2000; Claessens et al., 2003; Bellavista et al., 2004). In this sense, a line of future research is to incorporate proper security mechanisms into the agent execution model of GRATIS, and also to study how these mechanisms can be integrated with existing security services for the Grid such as GSI[1].

[1]The Grid Security Infrastructure (GSI): http://www.globus.org/security/overview.html

8.3.6 Integration with Mobile Devices

Mobile devices such as Personal Digital Assistant (PDA), cell and smart phones, and wearable computers, are becoming increasingly common. As time goes on, the trend seems to significantly increase, which will result in the following years in an unprecedented, huge mobile computing community. Yet, these devices are often limited in terms of resource capabilities, since processing power is low, battery life is finite, and memory space and storage capacity are heavily constrained. In addition, mobile devices commonly are connected to the Internet through wireless links which are unreliable and slow. These facts limits the potential of mobile devices as elements fully integrated to the Grid for accessing and consuming its services from anywhere and everywhere.

In this context, mobile agent technology is particularly useful (Vasiu and Mahmoud, 2004; Spyrou et al., 2004). For example, agents can travel from a resource-constrained device to a wired network, and then locally perform an expensive computation by making use of the resources and services hosted at the new execution environment. Consequently, future investigations will also study how JGRIM agents can help mobile users to easily, transparently and efficiently leverage the services provided by the Grid. As an aside, as both mobile devices can be viewed as both Grid clients and Grid nodes hosting limited nevertheless useful and potentially idle resources (Chu and Humphrey, 2004), another research line is concerned with extending GRATIS and JGRIM concepts to Grids composed of static as well as mobile nodes.

8.3.7 Semantic Grids

One major problem of current Grids is that its metadata – that is, the information about the characteristics and capabilities of both Grid resources and services – is generated and used in an ad-hoc fashion, and is mostly hardwired in the middleware code libraries and database schemas (Corcho et al., 2006). This leads to Grid systems and applications that are brittle when faced with frequent changes in the syntactic structure of resource and service metadata, and the protocols to access it. Precisely, the Semantic Grid is an extension of the current Grid in which resources and services are associated well-defined meaning through machine-processable descriptions that maximize the potential for automatic sharing and reuse, thus better enabling computers and people to work in cooperation.

The Semantic Grid is essentially the application of the principles of the Semantic Web (Berners-Lee et al., 2001) to the Grid environment. As a matter of fact, the Semantic Grid is often viewed as the result of combining Grids and Semantic Web technologies in order to simultaneously achieve increased levels of computational performance and application interoperability. In this sense, future research will study how to enhance GRATIS metaservices and gridified applications with metadata processing capabilities. A preliminary result in this line includes Apollo (Mateos et al., 2006b; Mateos et al., 2006a), a semantic-based service publication and discovery system that is designed to be used in massively distributed computing environments such as the Web. As a first step, the research will be focused on applying and adapting the ideas underpinning Apollo to the JGRIM middleware.

8.4 Final remarks

The Grid is an arrangement of distributed resources such as processing power, storage, services and applications that aims at allowing users to improve the execution capabilities of their applications by many orders of magnitude. Just like the Web, which is a system offering a huge amount of information in the form of text, images and video that can be easily accessed by users through a Web browser, a major goal of the Grid Computing paradigm is to enable developers to benefit from its resources with minimum (or ideally zero) implementation effort. However, the goal seems to be very far from being achieved, as plugging applications onto a Grid setting still requires these applications to be significantly modified or carefully configured before they can efficiently take advantage of Grid resources.

In this light, the thesis outlined throughout this work addressed the problem of ease gridification of conventional applications, that is, porting to the Grid applications that were not originally thought to run on a Grid environment. GRATIS showed that it is possible to simultaneously gridifying software with less effort, producing efficient Grid applications, and preserve the original application logic. Essentially, GRATIS is a step towards providing gridification techniques in which applications are gridified in a fully transparent and effortless way. In summary, GRATIS is an important contribution to the area of Grid Computing that leaves the door open to future research so as to improve the scalability, deployability and security of gridified applications, thus promoting a broad adoption of the approach to build real and production Grid applications.

Appendix A

Source Code of the k-NN Application

This appendix contains the source code associated to the implementations of the k-NN algorithm discussed in Section 7.2. Particularly, Section A.1 shows the code of the original (i.e. non--gridified) application. Later, Sections A.2, A.3 and A.4 shows the code of the Ibis, JGRIM and ProActive implementations, respectively. For the sake of clarity, only the most relevant portions of the application code are shown in each case.

A.1 Original

```
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;

public class KNNAlgorithm {
  DatasetReader dataset;

  KNNAlgorithm() {
    dataset = new DatasetReader();
  }
  protected Instance[] getInstances(int item) {
    return dataset.readRange(item, item + 1000);
  }
  public double classifyInstance(Instance instance, int k)
    throws Exception {
    Double greatestClass = new Double(0.0);
    if (instance.hasMissingValue()) {
      throw new RuntimeException("No support for missing values!");
    }
```

```
    LinkedList neighborList = new LinkedList();
    int numInstances = dataset.numInstances();
    for (int i = 0; i < numInstances;) {
      Instance[] block = getInstances(i);
      for (int arrayIndex = 0; arrayIndex < 1000; arrayIndex++) {
        Instance trainInstance = block[arrayIndex];
        // Use "trainInstance" to update the current neighbor list
        // associated to "instance"
        . . .
      }
      i += 1000;
    }
    // Compute the most frequent class label into "greatestClass"
    return greatestClass.doubleValue();
  }
  public double[] classifyInstances(Instance[] instances, int k)
    throws Exception {
    double[] result = new double[instances.length];
    for (int i = 0; i < instances.length; ++i) {
      result[i] = classifyInstance(instances[i], k);
    }
    return result;
  }
}
```

A.2 Ibis

```
import ibis.satin.SatinObject;
import java.io.Serializable;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;

public class KNNAlgorithm extends SatinObject
  implements KNNSpawnInterface, Serializable {
  transient IbisDatasetClient dataset;

  KNNAlgorithm() {
  }
  protected Instance[] getInstances(int startIndex) {
    return dataset.readRange(startIndex, startIndex + 1000);
```

```
  }
  public double classifyInstance(Instance instance, int k)
    throws Exception {
    Double greatestClass = new Double(0.0);
    dataset = new IbisDatasetClient();
    if (instance.hasMissingValue()) {
      throw new RuntimeException("No support for missing values!");
    }
    LinkedList neighborList = new LinkedList();
    int numInstances = dataset.numInstances();
    for (int i = 0; i < numInstances;) {
      Instance[] block = getInstances(i);
      for (int arrayIndex = 0; arrayIndex < 1000; arrayIndex++) {
        Instance trainInstance = block[arrayIndex];
        // Use "trainInstance" to update the current neighbor list
        // associated to "instance"
        . . .
      }
      i += 1000;
    }
    // Compute the most frequent class label into "greatestClass"
    return greatestClass.doubleValue();
  }
  public double[] classifyInstances(Instance[] instances, int k)
    throws Exception {
    ValueHolder holder = new ValueHolder(instances.length);
    holder.exportObject();
    for (int i = 0; i < instances.length; ++i) {
      spawn_classify(holder, instances[i], k, i);
    }
    sync();
    return holder.getResults();
  }
  public void spawn_classify(ValueHolder holder, Instance i,
    int k, int child) {
    try {
      double value = this.classifyInstance(i, k);
      holder.setValue(child, value);
    } catch (Exception e) {
      e.printStackTrace();
    }
```

```
  }
}

import ibis.satin.Spawnable;

public interface KNNSpawnInterface extends Spawnable {
  public void spawn_classify(ValueHolder holder, Instance i,
                             int k, int child);
}

import ibis.satin.SharedObject;

final class ValueHolder extends SharedObject
  implements WriteInterface {
  double[] array = null;

  public ValueHolder(int instances) {
    array = new double[instances];
  }
  public void setValue(int child, double value) {
    array[child] = value;
  }
  public double[] getResults() {
    return array;
  }
}

import ibis.satin.WriteMethodsInterface;

public interface WriteInterface extends WriteMethodsInterface {
  public void setValue(int child, double value);
}
```

A.3 JGRIM

```
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;

public class KNNAlgorithm extends core.JGRIMAgent {
  DatasetInterface dataset;
```

```
ParallelMethodInterface selfdependency;

public KNNAlgorithm() {
}
public void setdataset(DatasetInterface dataset) {
  this.dataset = dataset;
}
public DatasetInterface getdataset() {
  return dataset;
}
public ParallelMethodInterface getselfdependency() {
  return selfdependency;
}
public void setselfdependency(
  ParallelMethodInterface selfdependency) {
  this.selfdependency = selfdependency;
}
protected Instance[] getInstances(int item) {
  return getdataset().readRange(item, item + 1000);
}
public double classifyInstance(Instance instance, int k)
  throws Exception {
  Double greatestClass = new Double(0.0);
  if (instance.hasMissingValue()) {
    throw new RuntimeException("No support for missing values!");
  }
  LinkedList neighborList = new LinkedList();
  int numInstances = getdataset().numInstances();
  for (int i = 0; i < numInstances;) {
    Instance[] block = getInstances(i);
    for (int arrayIndex = 0; arrayIndex < 1000; arrayIndex++) {
      Instance trainInstance = block[arrayIndex];
      // Use "trainInstance" to update the current neighbor list
      // associated to "instance"
      . . .
    }
    i += 1000;
  }
  // Compute the most frequent class label into "greatestClass"
  return greatestClass.doubleValue();
}
```

```
  public double[] classifyInstances(Instance[] instances, int k,
    int start, int end) throws Exception {
    if (end - start == 1)
      return new double[] { classifyInstance(instances[start], k) };
    int mid = (start + end) / 2;
    double[] res1 = classifyInstances(instances, k, start, mid);
    double[] res2 = classifyInstances(instances, k, mid, end);
    double[] result = new double[res1.length + res2.length];
    System.arraycopy(res1, 0, result, 0, res1.length);
    System.arraycopy(res2, 0, result, res1.length, res2.length);
    return result;
  }
  public double[] classifyInstances(Instance[] instances, int k)
    throws Exception {
    return getselfdependency().classifyInstances(instances, k,
                                          0, instances.length);
  }
}

public interface DatasetInterface {
  public int numInstances();
  public Instance readItem(int itemNumber);
  public Instance[] readRange(int start, int end);
}

public interface ParallelMethodInterface {
  public double[] classifyInstances(Instance[] instance,
    int k, int start, int end) throws Exception;
}

import ibis.satin.SatinObject;
import java.io.ByteArrayInputStream;
import java.io.ObjectInputStream;
import java.lang.reflect.Method;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.zip.GZIPInputStream;

public class KNNAlgorithmPeer extends SatinObject
  implements ParallelMethodInterfacePeer {
```

```
  private byte[] serializedAgent = null;
  private transient Object targetAgent = null;

  public KNNAlgorithmPeer() {
  }
  public void setSerializedAgent(byte[] serializedAgent) {
    this.serializedAgent = serializedAgent;
  }
  public Object getTargetAgent() throws Throwable {
    if (targetAgent == null) {
      ByteArrayInputStream bStream = new ByteArrayInputStream(
                                                    serializedAgent);
      GZIPInputStream zStream = new GZIPInputStream(bStream);
      ObjectInputStream oStream = new ObjectInputStream(zStream);
      targetAgent = oStream.readObject();
    }
    return targetAgent;
  }
  protected Object getDependency(String name) {
    try {
      Object agent = getTargetAgent();
      String mname = "get" + name;
      Method m = agent.getClass().getMethod(mname, new Class[] {});
      Object res = m.invoke(agent, new Object[] {});
      return res;
    } catch (Throwable e) {
      e.printStackTrace();
      return null;
    }
  }
  public DatasetInterface getdataset() {
    return (DatasetInterface) getDependency("dataset");
  }
  public double[] classifyInstances(Instance[] instances, int k,
    int start, int end) throws Exception {
    // Same implementation as the KNNAlgorithm class, but properly
    // including the sync() Ibis primitive
  }
  public double[] classifyInstances(Instance[] instances, int k)
    throws Exception {
```

```
    return classifyInstances(instances, k, 0, instances.length);
}
}

import ibis.satin.Spawnable;

public interface ParallelMethodInterfacePeer extends Spawnable {
  public double[] classifyInstances(Instance[] instance,
    int k, int start, int end) throws Exception;
}
```

A.4 ProActive

```
import java.io.Serializable;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Vector;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.core.util.wrapper.GenericTypeWrapper;

public class KNNAlgorithm implements Serializable {
  transient ProactiveDatasetClient dataset;
  Node[] nodes = null;
  private boolean[] isNodeAvailable;
  private Hashtable assignedWork;
  Hashtable childNodeHash;

  public KNNAlgorithm() {
  }
  public KNNAlgorithm(Node[] nodes) {
    this.nodes = nodes;
    initializeNodes(nodes.length);
    assignedWork = new Hashtable();
    childNodeHash = new Hashtable();
  }
  protected void initializeNodes(int numberOfNodes) {
    this.isNodeAvailable = new boolean[numberOfNodes];
    for (int i = 0; i < numberOfNodes; i++)
      isNodeAvailable[i] = true;
```

```
  }
  protected int getAvailableNode() {
    for (int i = 0; i < isNodeAvailable.length; i++)
      if (isNodeAvailable[i])
        return i;
    return -1;
  }
  protected Vector getCurrentFutureList() {
    Vector futureList = new Vector();
    Enumeration currentFl = assignedWork.elements();
    while (currentFl.hasMoreElements())
      futureList.add(currentFl.nextElement());
    return futureList;
  }
  protected ProactiveDatasetClient getDataset() {
    if (dataset == null) {
      dataset = new ProactiveDatasetClient();
    }
    return dataset;
  }
  protected ProactiveDatasetClient getDataset() {
    if (dataset == null) {
      dataset = new ProactiveDatasetClient();
    }
    return dataset;
  }
  protected Instance[] getInstances(int item) {
    return getDataset().readRange(item, item + 1000);
  }
  public GenericTypeWrapper classifyInstance(Instance instance, int k,
    int instanceNumber) throws Exception {
    Double greatestClass = new Double(0.0);
    if (instance.hasMissingValue()) {
      throw new RuntimeException("No support for missing values!");
    }
    LinkedList neighborList = new LinkedList();
    int numInstances = getDataset().numInstances();
    for (int i = 0; i < numInstances;) {
      Instance[] block = getInstances(i);
      for (int arrayIndex = 0; arrayIndex < 1000; arrayIndex++) {
        Instance trainInstance = block[arrayIndex];
```

```
        // Use "trainInstance" to update the current neighbor list
        // associated to "instance"
        . . .
      }
      i += 1000;
    }
    // Compute the most frequent class label into "greatestClass"
    TaskResult result = new TaskResult(
              greatestClass.doubleValue(), instanceNumber);
    return new GenericTypeWrapper(result);
  }
  public double[] classifyInstances(Instance[] instances, int k)
    throws Exception {
    GenericTypeWrapper[] temp =
                    new GenericTypeWrapper[instances.length];
    for (int i = 0; i < instances.length; ++i) {
      int avail = getAvailableNode();
      if (avail == -1) {
        Vector fl = getCurrentFutureList();
        int finished = ProActive.waitForAny(fl);
        GenericTypeWrapper element =
                    (GenericTypeWrapper) fl.elementAt(finished);
        int solvedChild = ((TaskResult) element.getObject())
                                                .getChild();
        assignedWork.remove(solvedChild);
        int node = childNodeHash.remove(solvedChild);
        isNodeAvailable[node] = true;
        avail = node;
      }
      KNNAlgorithm worker = (KNNAlgorithm) ProActive.newActive(
                    getClass().getName(), null, nodes[avail]);
      temp[i] = worker.classifyInstance(instances[i], k, i);
      isNodeAvailable[avail] = false;
      assignedWork.put(i, temp[i]);
      childNodeHash.put(i, avail);
    }
    double[] result = new double[instances.length];
    for (int i = 0; i < instances.length; ++i) {
      result[i] = ((TaskResult) temp[i].getObject())
                                        .getInstanceClass();
    }
```

```
    return result;
  }
}

import java.io.Serializable;

public class TaskResult implements Serializable {
  private double instanceClass;
  private int child;

  public TaskResult(double instanceClass, int child) {
    this.instanceClass = instanceClass;
    this.child = child;
  }
  public int getChild() {
    return child;
  }
  public double getInstanceClass() {
    return instanceClass;
  }
}
```

Appendix B

Source Code of the Restoration Application

This appendix includes the source code associated to the implementations of the image restoration application presented in Section 7.3. Specifically, Section B.1 shows the code of the original (i.e. non-Grid aware) restoration application. Later, Sections B.2, B.3 and B.4 shows the code of the Ibis, JGRIM and ProActive implementations, respectively. For simplicity reasons, the most relevant portions of the code of the master and worker components are shown in each case.

B.1 Original

```
import java.io.IOException;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import utils.ftp.FTPClientImpl;

public class ImageRestorer implements Master {
  private int iterations;
  private int children;
  private Object[] results = null;
  private int currentChild = 0;
  private int remaining = 0;
  private FTPClientImpl client = null;

  public ImageRestorer(int iterations, int children) {
    this.iterations = iterations;
    this.children = children;
```

```
    this.remaining = children;
    this.client = new FTPClientImpl();
    this.results = new Object[children];
  }

  public void restoreImage(String fileLocation, String fileName)
    throws Exception {
    PlanarImage image = getImage(fileLocation, fileName);
    int width = image.getWidth();
    int height = image.getHeight();
    Vector v = RestoreUtils.createTiles(image,
                              width / children, height);
    for (Iterator iter = v.iterator(); iter.hasNext();) {
      byte[] subImagePixels = (byte[]) iter.next();
      Hashtable data = new Hashtable();
      data.put("pixels", subImagePixels);
      data.put("iterations", iterations);
      submitWork(data, currentChild);
      currentChild++;
    }
    waitAll();
    RestoreUtils.mergeSubImages(getResults(), width / children,
                                height, "/tmp/res_" + fileName);
  }
  protected PlanarImage getImage(String fileLocation, String fileName)
    throws IOException {
    client.transfer(fileLocation, fileName, "/tmp",
                    System.getProperty("imageftp.user"),
                    System.getProperty("imageftp.passwd"));
    return JAI.create("fileload", "/tmp/" + fileName);
  }
  public void submitWork(Hashtable taskData, int currentChild) {
    Worker worker = new Worker(taskData, this, currentChild);
    startWorker(worker);
  }
  public void startWorker(Worker worker) {
    Thread thr = new Thread(worker);
    thr.start();
  }
  public void receiveWork(Object result, int child) {
    results[child] = result;
```

```
    synchronized (this) {
      notify();
      remaining--;
    }
  }
  public void waitAll() {
    while (getRemaining() > 0) {
      synchronized (this) {
        try {
          wait();
        } catch (InterruptedException ie) {
          ie.printStackTrace();
        }
      }
    }
  }
  public synchronized int getRemaining() {
    return remaining;
  }
  public Object[] getResults() {
    return results;
  }
}

import java.util.Hashtable;

public interface Master {
  public void submitWork(Hashtable taskData, int child);
  public void receiveWork(Object result, int child);
  public void waitAll();
}

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.RandomAccessFile;
import java.util.Hashtable;

public class Worker implements Runnable {
  private Hashtable taskData = null;
```

```
  private Master master = null;
  private int child = 0;

  public Worker(Hashtable taskData, Master master, int child) {
    this.taskData = taskData;
    this.master = master;
    this.child = child;
  }
  public void run() {
    byte[] imageData = (byte[]) taskData.get("pixels");
    int iterations = (Integer) taskData.get("iterations");
    byte[] result = null;
    try {
      result = restoreImage(imageData, iterations);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    master.receiveWork(result, child);
  }
  protected byte[] restoreImage(byte[] imageData, int iterations)
    throws Exception {
    // Runs the actual restoration process on "imageData"
  }
}
```

B.2 Ibis

```
import ibis.satin.SatinObject;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Vector;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import utils.ftp.FTPClientImpl;

public class ImageRestorer extends SatinObject
             implements ImageRestorerSpawnInterface {
  private int iterations;
  private int children;
  private Object[] results = null;
```

```
  private int currentChild = 0;
  private FTPClientImpl client = null;

  public ImageRestorer(int iterations, int children) {
    this.iterations = iterations;
    this.children = children;
    this.results = new Object[children];
    this.client = new FTPClientImpl();
  }
  public void restoreImage(String fileLocation, String fileName)
    throws Exception {
    PlanarImage image = getImage(fileLocation, fileName);
    int width = image.getWidth();
    int height = image.getHeight();
    Vector v = RestoreUtils.createTiles(image,
                              width / children, height);
    submitWork(v);
    RestoreUtils.mergeSubImages(getResults(), width / children,
                                height, "/tmp/res_" + fileName);
  }
  public void submitWork(Vector tiles) {
    Vector temp = submitWork(tiles, 0);
    sync();
    this.results = temp.toArray();
  }
  public Vector submitWork(Vector tiles, int currentChild) {
    if (tiles.size() == 1) {
      byte[] subImagePixels = (byte[]) tiles.firstElement();
      Hashtable data = new Hashtable();
      data.put("pixels", subImagePixels);
      data.put("iterations", iterations);
      Worker worker = new Worker(data, currentChild);
      Vector result = new Vector();
      result.addElement(worker.run());
      return result;
    }
    int mid = tiles.size() / 2;
    Vector tilesStart = grabElements(tiles, 0, mid);
    Vector tilesEnd = grabElements(tiles, mid, tiles.size());
    Vector res1 = submitWork(tilesStart, currentChild);
    Vector res2 = submitWork(tilesEnd, currentChild + mid);
```

```
    sync();
    res1.addAll(res2);
    return res1;
  }
  protected Vector grabElements(Vector target, int start, int end) {
    Vector result = new Vector();
    for (; start < end; start++)
      result.addElement(target.elementAt(start));
    return result;
  }
}

import ibis.satin.Spawnable;
import java.util.Vector;

public interface ImageRestorerSpawnInterface extends Spawnable {
  public Vector submitWork(Vector tiles, int currentChild);
}

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.RandomAccessFile;
import java.io.Serializable;
import java.util.Hashtable;

public class Worker implements Serializable {
  private Hashtable taskData = null;
  private int child = 0;

  public Worker(Hashtable taskData, int child) {
    this.taskData = taskData;
    this.child = child;
  }
  public byte[] run() {
    byte[] imageData = (byte[]) taskData.get("pixels");
    int iterations = (Integer) taskData.get("iterations");
    byte[] result = null;
    try {
      result = restoreImage(imageData, iterations);
```

```
    } catch (Exception e) {
      e.printStackTrace();
    }
    return result;
  }
  protected byte[] restoreImage(byte[] imageData, int iterations)
    throws Exception {
    // Runs the actual restoration process on "imageData"
  }
}
```

B.3 JGRIM

```
import java.io.IOException;
import java.util.Hashtable;
import java.util.Vector;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import utils.ftp.FTPClientInterface;
import core.JGRIMAgent;

public class ImageRestorer extends JGRIMAgent {
  private int iterations;
  private int children;
  private Object[] results = null;
  private FTPClientInterface client = null;
  String image = null;
  ParallelMethodInterface selfdependency;

  public ImageRestorer(int iterations, int children, String image) {
    this.iterations = iterations;
    this.children = children;
    this.results = new Object[children];
    this.image = image;
  }
  public void restoreImage(String fileLocation, String fileName)
    throws Exception {
    // Same implementation as the Ibis version, with the exception
    // of the explicit calls to the sync() Ibis primitive
  }
  public void submitWork(Vector tiles) {
```

```
    Vector temp = getselfdependency().submitWork(tiles, 0);
    this.results = temp.toArray();
  }
  protected PlanarImage getImage(String fileLocation, String fileName)
    throws IOException {
    getclient().transfer(fileLocation, fileName, "/tmp",
                         System.getProperty("imageftp.user"),
                         System.getProperty("imageftp.passwd"));
    return JAI.create("fileload", "/tmp/" + fileName);
  }
  public Vector submitWork(Vector tiles, int currentChild) {
    // Same implementation as the Ibis version, with the exception
    // of the explicit calls to the sync() Ibis primitive
  }
  public FTPClientInterface getclient() {
    return client;
  }
  public void setclient(FTPClientInterface client) {
    this.client = client;
  }
  public ParallelMethodInterface getselfdependency() {
    return selfdependency;
  }
  public void setselfdependency(
    ParallelMethodInterface selfdependency) {
    this.selfdependency = selfdependency;
  }
  public void setResults(Object[] results) {
    this.results = results;
  }
  public int getChildren() {
    return children;
  }
  public void setChildren(int children) {
    this.children = children;
  }
  public int getIterations() {
    return iterations;
  }
  public void setIterations(int iterations) {
    this.iterations = iterations;
```

```
  }
  public void run() {
    try {
      restoreImage("ftp://VPNServer", image);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}

import java.io.IOException;

public interface FTPClientInterface {
  public void transferAnon(String ftpLocation, String targetFile,
                           String destDir) throws IOException;
  public void transfer(String ftpLocation, String targetFile,
                       String destDir, String username,
                       String password) throws IOException;
  public long getFileSize(String ftpLocation, String targetFile,
            String username, String password) throws IOException;
  public long getFileSizeAnon(String ftpLocation,
                           String targetFile) throws IOException;
}

import java.util.Vector;

public interface ParallelMethodInterface {
  public Vector submitWork(Vector tiles, int currentChild);
}

import java.io.Serializable;

public class Worker implements Serializable {
  // Same implementation as the Ibis version
}

import ibis.satin.SatinObject;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.lang.reflect.Method;
```

```
import java.util.Hashtable;
import java.util.Vector;
import java.util.zip.GZIPInputStream;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import utils.ftp.FTPClientInterface;

public class ImageRestorerPeer extends SatinObject
  implements ParallelMethodInterfacePeer {
  private int iterations;
  private int children;
  private Object[] results = null;
  private byte[] serializedAgent = null;
  private transient Object targetAgent = null;

  public ImageRestorerPeer() {
  }
  public ImageRestorerPeer(int iterations, int children) {
    this.iterations = iterations;
    this.children = children;
    this.results = new Object[children];
  }
  public void setSerializedAgent(byte[] serializedAgent) {...}
  public Object getTargetAgent() throws Throwable {...}
  protected Object getDependency(String name) {...}
  public void restoreImage(String fileLocation, String fileName)
    throws Exception {...}
  public void submitWork(Vector tiles) {
    // Same implementation as the ImageRestorer class, but properly
    // including the sync() Ibis primitive
  }
  protected PlanarImage getImage(String fileLocation, String fileName)
    throws IOException {...}
  public Vector submitWork(Vector tiles, int currentChild) {
    // Same implementation as the ImageRestorer class, but properly
    // including the sync() Ibis primitive
  }
  public FTPClientInterface getClient() {
    return (FTPClientInterface) getDependency("client");
  }
  // Setters/getters for the results, children and
```

```
  // iterations instance variables
  ...
}

import ibis.satin.Spawnable;
import java.util.Vector;

public interface ParallelMethodInterfacePeer extends Spawnable {
  public Vector submitWork(Vector tiles, int currentChild);
}
```

B.4 ProActive

```
import java.io.IOException;
import java.io.Serializable;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.core.util.wrapper.GenericTypeWrapper;
import utils.ftp.FTPClientImpl;

public class ImageRestorer implements Master, Serializable {
  private int iterations;
  private int children;
  private Object[] results = null;
  private int currentChild = 0;
  private FTPClientImpl client = null;
  private Node[] nodes = null;
  private Vector futureList = null;
  // Data structures to maintain the association
  // between workers and their tasks
  ...
```

```
public ImageRestorer() {
}
public ImageRestorer(int iterations, int children, Node[] nodes) {
  this.iterations = iterations;
  this.children = children;
  this.client = new FTPClientImpl();
  this.results = new Object[children];
  this.nodes = nodes;
  this.futureList = new Vector();
  // Initialize the above data structures
  ...
}
public void restoreImage(String fileLocation, String fileName)
  throws Exception {
  PlanarImage image = getImage(fileLocation, fileName);
  int width = image.getWidth();
  int height = image.getHeight();
  Vector v = RestoreUtils.createTiles(image,
                          width / children, height);
  for (Iterator iter = v.iterator(); iter.hasNext();) {
    byte[] subImagePixels = (byte[]) iter.next();
    Hashtable data = new Hashtable();
    data.put("pixels", subImagePixels);
    data.put("iterations", iterations);
    submitWork(data, currentChild);
    currentChild++;
  }
  waitAll();
  RestoreUtils.mergeSubImages(getResults(), width / children,
                            height, "/tmp/res_" + fileName);
}
public void submitWork(Hashtable taskData, int currentChild) {
  // Task management/active object deployment is implemented
  // similar to the ProActive version of the k-NN application
  // (see Appendix A)
}
public void startWorker(Worker worker) {
  GenericTypeWrapper wrapper = worker.run();
  futureList.addElement(wrapper);
}
public void receiveWork(Object result, int child) {
```

```
    results[child] = result;
  }
  public void waitAll() {
    ProActive.waitForAll(futureList);
  }
  public Object[] getResults() {
    return results;
  }
}

import java.util.Hashtable;

public interface Master {
  public void submitWork(Hashtable taskData, int child);
  public void receiveWork(Object result, int child);
  public void waitAll();
}

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.RandomAccessFile;
import java.util.Hashtable;
import org.objectweb.proactive.core.util.wrapper.GenericTypeWrapper;

public class Worker implements Runnable, Serializable {
  private Hashtable taskData = null;
  private Master master = null;
  private int child = 0;

  public Worker() {
  }
  public Worker(Hashtable taskData, Master master, int child) {
    this.taskData = taskData;
    this.master = master;
    this.child = child;
  }
  public GenericTypeWrapper run() {
    byte[] imageData = (byte[]) taskData.get("pixels");
    int iterations = (Integer) taskData.get("iterations");
```

```
    byte[] result = null;
    try {
      result = restoreImage(imageData, iterations);
    } catch (Exception e) {
      e.printStackTrace();
    }
    master.receiveWork(result, child);
    return new GenericTypeWrapper();
  }
  protected byte[] restoreImage(byte[] imageData, int iterations)
    throws Exception {
    // Runs the actual restoration process on "imageData"
  }
}
```

Appendix C

Result Tables of the k-NN Application

This appendix presents the data tables containing the measurements of performance and network resources usage associated to the experiments that were carried out on the k-NN algorithm presented in Section 7.2. Sections C.1, C.2, C.3 and C.4 shows the result tables of the Ibis, JGRIM, JGRIM (caching policy) and ProActive implementations, respectively.

C.1 Ibis

C.1.1 Total Execution Time (milliseconds)

Run	5 inst.	10 inst.	15 inst.	20 inst.	25 inst.
1	121128	126420	127544	201262	215567
2	118697	94330	118838	170997	196996
3	88066	90984	138189	171922	183621
4	115203	89197	134001	166977	178314
5	108066	113827	104118	170547	178045
6	88714	89567	111675	169437	177414
7	87902	91078	106656	166094	178341
8	87614	91123	105805	171677	174843
9	114740	119075	118056	171797	175750
10	88750	115651	138758	170355	194435
Std. Dev. (ms)	14794.58	14720.51	13501.24	10092.22	13130.76
Average (ms)	101888	102125.2	120364	173106.5	185332.6

Std. Dev./Average	0.15	0.14	0.11	0.06	0.07
Average (min)	1.70	1.70	2.01	2.89	3.09
Total elapsed time (min)	16.98	17.02	20.06	28.85	30.89

C.1.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	35993009	2099802	345354610	3176969	35477677	2032936
10	119419067	1066590	582141183	1306173	107784688	984417
15	155174428	1279633	599960330	1730737	305902731	1084436
20	238266560	1792497	840311771	2460683	341259992	1598266
25	238722086	1364148	1196190671	1520513	457189048	1098144
Subtotal (MB)	751.09	7.25	3398.86	9.72	1189.82	6.48
Total cluster (MB)	758.34		3408.58		1196.30	
Total traffic (MB)	5363.22 (extra: 23.46 MB)					

C.1.3 TCP Packets

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	37540	26083	391604	43104	36536	28214
10	118376	11890	579278	17201	108039	13388
15	153001	14356	613952	22984	293540	14548
20	235047	20451	861012	33123	328283	21845
25	235351	14419	1170024	19827	439478	14631
Subtotal	779315	87199	3615870	136239	1205876	92626

Total cluster	866514	3752109	1298502
Total packets	5917125 (extra: 316064)		

C.2 JGRIM

C.2.1 Total Execution Time (milliseconds)

Run	5 inst.	10 inst.	15 inst.	20 inst.	25 inst.
1	131115	131201	157449	216008	230945
2	134491	123996	129689	205018	216253
3	119578	121733	131911	206231	212337
4	111770	127116	132335	200286	211657
5	118659	115692	124226	200802	205518
6	119336	120352	133063	201718	209521
7	113074	118245	130029	198214	215020
8	117552	116230	126613	196592	210272
9	110851	123145	130485	193185	211263
10	114640	123187	128144	191695	205731
Std. Dev. (ms)	7902.86	4800.77	9208.63	7028.95	7227.19
Average (ms)	119106.6	122089.7	132394.4	200974.9	212851.7
Std. Dev./Average	0.07	0.04	0.07	0.03	0.03
Average (min)	1.99	2.03	2.21	3.35	3.55
Total elapsed time (min)	19.85	20.35	22.07	33.5	35.48

C.2.2 Network Traffic (bytes)

	Ale cluster		ISISTAN cluster		Bianca cluster	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	47921726	2394527	291906922	3068005	23671388	1751910
10	119332748	1771121	599489310	1816926	117680350	846962
15	238624266	1480555	600516073	2000219	234640220	875019
20	250788230	2973686	830277453	3881903	352945382	2100438
25	358022598	2267393	1197779022	2826101	376395959	1465356
Subtotal (MB)	967.68	10.38	3356.90	12.96	1054.13	6.71
Total cluster (MB)	978.07		3369.87		1060.84	
Total traffic (MB)	5408.78 (extra: 30.06 MB)					

C.2.3 TCP Packets

	Ale cluster		ISISTAN cluster		Bianca cluster	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	48630	23194	336093	37780	24649	23920
10	117712	11780	589391	15649	113292	10993
15	233874	8552	601520	14176	223790	9049
20	247398	26378	858746	40979	336645	24786
25	351419	14328	1177025	20791	359801	15208
Subtotal	999033	84232	3562775	129375	1058177	83956
Total cluster	1083265		3692150		1142133	
Total packets	5917548 (extra: 297563)					

C.3 JGRIM (caching policy)

C.3.1 Total Execution Time (milliseconds)

Run	5 inst.	10 inst.	15 inst.	20 inst.	25 inst.
1	134621	134252	137703	202517	213880
2	109949	106883	105597	181040	181946
3	105506	107930	116730	179115	183749
4	105846	112135	113480	179622	184269
5	104914	107247	111243	190870	185318
6	104238	105271	111214	177909	184723
7	110868	105978	114609	182561	184113
8	107677	108215	109302	170865	185374
9	109138	108844	105098	176058	186596
10	107466	108475	119720	179990	186031
Std. Dev. (ms)	8919.55	8543.68	9355.51	8786.86	9325.00
Average (ms)	110022.3	111323.0	114469.6	182054.7	187599.9
Std. Dev./Average	0.08	0.08	0.08	0.05	0.05
Average (min)	1.83	1.86	1.91	3.03	3.13
Total elapsed time (min)	18.34	18.42	19.08	30.34	31.27

C.3.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	12193213	2236744	62637919	2466842	12006335	1219457
10	12116868	1619244	62074000	1706405	11905327	810227
15	12079335	1444770	62084130	1938815	11905423	663558
20	12200954	2868660	65943298	3924059	11973100	1681626

25	12074182	2009275	63853807	2382516	11872865	1282561
Subtotal (MB)	57.85	9.71	301.93	11.84	56.90	5.40
Total cluster (MB)	67.56		313.77		62.29	
Total traffic (MB)	443.63 (extra: 26.95 MB)					

C.3.3 TCP Packets

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	13095	17730	100271	27815	12810	17352
10	13547	11330	70562	14817	12957	10031
15	12641	7973	72879	11542	12226	6887
20	13935	23376	119820	36701	13372	21935
25	13784	13671	73911	18408	13666	12883
Subtotal	67002	74080	437443	109283	65031	69088
Total cluster	141082		546726		134119	
Total packets	821927 (extra: 252451)					

C.4 ProActive

C.4.1 Total Execution Time (seconds)

Run	**5 inst.**	**10 inst.**	**15 inst.**	**20 inst.**	**25 inst.**
1	133	171	201	216	248
2	133	167	178	218	259
3	135	166	183	219	259
4	130	165	180	236	257
5	130	168	205	237	257

6	132	164	173	216	259
7	137	173	176	217	256
8	132	165	175	217	258
9	132	165	175	232	256
10	136	167	183	253	257
Std. Dev. (secs)	2.36	2.88	11.15	12.74	3.24
Average (secs)	133.0	167.1	182.9	226.1	256.6
Std. Dev./Average	0.02	0.02	0.06	0.06	0.01
Average (min)	2.22	2.79	3.05	3.77	4.28
Total elapsed time (min)	22.17	27.85	30.48	37.68	42.77

C.4.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	119724820	1746741	0	1080899	117560112	443230
10	119744722	2306580	595306172	1344802	117569022	468437
15	239077678	2530941	595237375	1358298	235134264	547765
20	358441935	2913094	631053272	1472469	352693350	638433
25	358442628	3393550	1179864459	1633644	364352959	661672
Subtotal (MB)	1140.05	12.29	2862.42	6.57	1132.31	2.63
Total cluster (MB)	1152.35		2868.99		1134.94	
Total traffic (MB)	5156.27 (extra: 21.50 MB)					

C.4.3 TCP Packets

	Ale cluster		ISISTAN cluster		Bianca cluster	
Instances	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
5	119440	10706	0	8115	111528	3266
10	119626	12132	553255	9679	111693	3527
15	236070	12839	553846	9874	223316	4057
20	352809	14164	587014	10761	334825	4649
25	352774	15209	1108358	11632	345364	4876
Subtotal	1180719	65050	2802473	50061	1126726	20375
Total cluster	1245769		2852534		1147101	
Total packets	5245404 (extra: 135486)					

Appendix D

Result Tables of the Restoration Application

This appendix presents the data tables containing the measurements of performance and network resources usage associated to the experiments that were carried out on the restoration application presented in Section 7.3. Sections D.1, D.2, D.3 and D.4 shows the result tables of the Ibis, JGRIM, JGRIM (move policy) and ProActive implementations, respectively.

D.1 Ibis

D.1.1 Total Execution Time (milliseconds)

Run	Image 1	Image 2	Image 3	Image 4	Image 5
1	155746	382553	631631	839782	944597
2	158862	356035	607022	840158	896725
3	136198	349497	643679	879266	956731
4	163541	399742	637130	709532	1043207
5	188527	351618	627238	870509	988298
6	173809	392974	655232	825621	1054727
7	141200	367648	657021	869676	931633
8	149897	385438	596638	860451	1039956
9	171396	347119	598948	892861	872852
10	180653	348284	649935	788145	1040525
Std. Dev. (ms)	16916.63	20346.44	22646.24	54186.8	66132.16
Average (ms)	161982.9	368090.8	630447.4	837600.1	976925.1

Std. Dev./Average	0.10	0.06	0.04	0.06	0.07
Average (min)	2.70	6.13	10.51	13.96	16.28
Total elapsed time (min)	27.00	61.35	105.07	139.60	162.82
Average(i)/Average(i-1)	-	2.27	1.71	1.33	1.17
Image size (bytes)	445848	924794	1551932	1862822	2467004
Throughput (KB/min)	161.28	147.21	144.24	130.31	147.97

D.1.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	1044175	5733714	4681559	10568730	282706	3091175
Image 2	1590774	11829591	10148085	22249633	1081995	5348232
Image 3	796071	21050161	20789325	39843352	881673	9657627
Image 4	661322	26378506	30251104	48252813	1191294	12553535
Image 5	4761146	31261763	29107439	60756800	916425	15489082
Subtotal (MB)	8.44	91.79	90.58	173.26	4.15	44.00
Total cluster (MB)	100.24		263.83		48.15	
Total traffic (MB)	412.23 (25% intra,75% extra)					

D.1.3 TCP Packets

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	1907	20479	40903	31901	1626	17473
Image 2	2989	43195	96584	69268	4090	36615
Image 3	3395	76740	178524	125374	6103	66285

Image 4	4332	106950	272020	175932	8717	96238
Image 5	8721	117543	283633	192167	8067	100841
Subtotal	21344	364907	871664	594642	28603	317452
Total cluster	386251		1466306		346055	
Total packets	2198612 (42% intra, 58% extra)					

D.2 JGRIM

D.2.1 Total Execution Time (milliseconds)

Run	Image 1	Image 2	Image 3	Image 4	Image 5
1	149828	311194	551545	690227	854520
2	141567	303336	561653	688101	824640
3	144757	285123	572661	695953	778367
4	143988	301309	560217	754831	868289
5	134009	323744	533166	713058	876086
6	156129	324879	510969	759095	840967
7	141076	325700	577893	753706	874180
8	159246	289248	574144	730191	875516
9	147320	326324	540914	767413	856541
10	148905	321950	543824	699496	925772
Std. Dev. (ms)	7389.97	15707.48	21007.18	31462.96	38515.01
Average (ms)	146682.50	311280.70	552698.60	725207.10	857487.80
Std. Dev./Average	0.05	0.05	0.04	0.04	0.04
Average (min)	2.44	5.19	9.21	12.09	14.29
Total elapsed time (min)	24.45	51.88	92.12	120.87	142.91
Average(i)/Average(i-1)	-	2.12	1.78	1.31	1.18
Image size (bytes)	445848	924794	1551932	1862822	2467004
Throughput (KB/min)	178.1	174.08	164.53	150.51	168.58

D.2.2 Network Traffic (bytes)

	Ale cluster		ISISTAN cluster		Bianca cluster	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	794284	5485968	2507310	10165850	205316	1952224
Image 2	1729418	10984353	3927723	19475017	477456	4338607
Image 3	1725921	18549707	10334401	34136470	1013729	7486738
Image 4	1536463	22475407	9351611	41023730	1493963	10860664
Image 5	3651746	27036990	10701399	51956964	1550325	12051550
Subtotal (MB)	9.00	80.62	35.12	149.50	4.52	34.99
Total cluster (MB)	89.62		184.61		39.51	
Total traffic (MB)	313.74 (16% intra, 84% extra)					

D.2.3 TCP Packets

	Ale cluster		ISISTAN cluster		Bianca cluster	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	1290	13399	13393	18202	804	7917
Image 2	2448	24435	21652	31656	1704	14349
Image 3	2650	40169	39185	53890	3447	25383
Image 4	3356	53063	52538	69550	4153	33401
Image 5	5374	62459	58996	81741	4760	37532
Subtotal	15118	193525	185764	255039	14868	118582
Total cluster	208643		440803		133450	
Total packets	782896 (28% intra, 72% extra)					

D.3 JGRIM (move policy)

D.3.1 Total Execution Time (milliseconds)

Run	Image 1	Image 2	Image 3	Image 4	Image 5
1	111267	275184	405355	570176	663184
2	112878	267775	466274	553722	767989
3	113323	223339	476230	558828	698072
4	99582	242524	429368	549069	652632
5	125610	252888	479998	645923	756660
6	104279	230637	434709	605929	742243
7	101405	254760	468672	567304	634039
8	112071	273169	397285	531071	690436
9	104385	245264	395758	595253	629228
10	101481	268986	398702	649984	788898
Std. Dev. (ms)	7953.41	18022.53	35002.50	40581.23	58232.66
Average (ms)	108628.10	253452.60	435235.10	582725.90	702338.10
Std. Dev./Average	0.07	0.07	0.08	0.07	0.08
Average (min)	1.81	4.22	7.25	9.71	11.71
Total elapsed time (min)	18.10	42.24	72.54	97.12	117.06
Average(i)/Average(i-1)	-	2.33	1.72	1.34	1.21
Image size (bytes)	445848	924794	1551932	1862822	2467004
Throughput (KB/min)	240.49	213.80	208.93	187.31	205.81

D.3.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	148458	2339550	7681334	4186725	214857	2421402

Image 2	699491	5496225	11932234	8599605	317618	5622590
Image 3	1162919	9137274	22850725	15222437	1173487	7881943
Image 4	2022224	10507448	31347804	16995452	1639555	10353230
Image 5	941245	14467458	30670864	24080329	1430680	12574126
Subtotal (MB)	4.74	40.00	99.64	65.88	4.55	37.05
Total cluster (MB)	44.75		165.53		41.61	
Total traffic (MB)	251.88 (43% intra, 57% extra)					

D.3.3 TCP Packets

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	558	7722	19784	12617	769	8088
Image 2	1427	15585	38320	25224	1288	16359
Image 3	2468	25973	64374	41830	2664	25043
Image 4	3781	39676	116115	66781	4294	42787
Image 5	3231	43963	108886	72174	4084	42329
Subtotal	11465	132919	347479	218626	13099	134606
Total cluster	144384		566105		147705	
Total packets	858194 (43% intra, 57% extra)					

D.4 ProActive

D.4.1 Total Execution Time (seconds)

Run	**Image 1**	**Image 2**	**Image 3**	**Image 4**	**Image 5**
1	218	351	582	716	814
2	216	346	574	718	817

3	206	369	563	714	804
4	206	376	581	718	787
5	207	367	594	722	794
6	218	374	571	720	794
7	208	366	574	718	795
8	204	374	582	721	799
9	208	369	576	722	800
10	220	372	577	726	792
Std. Dev. (secs)	6.12	10.04	8.19	3.44	9.61
Average (secs)	211.1	366.4	577.4	719.5	799.6
Std. Dev./Average	0.029	0.027	0.014	0.005	0.012
Average (min)	3.52	6.11	9.62	11.99	13.33
Total elapsed time (min)	35.18	61.07	96.23	119.92	133.27
Average(i)/Average(i-1)	-	1.74	1.58	1.25	1.11
Image size (bytes)	445848	924794	1551932	1862822	2467004
Throughput (KB/min)	123.75	147.89	157.49	151.70	180.78

D.4.2 Network Traffic (bytes)

	Ale cluster		**ISISTAN cluster**		**Bianca cluster**	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	1883159	5920628	0	7519677	0	1730094
Image 2	3130948	8500154	0	13787557	0	2509867
Image 3	4906296	13051090	0	22370274	0	3425992
Image 4	5582678	14398938	0	26414632	0	3872121
Image 5	6731362	17675866	0	34172309	0	5576476
Subtotal (MB)	21.20	56.79	0	99.43	0	16.32
Total cluster (MB)	77.99		99.43		16.32	
Total traffic (MB)	193.75 (11% intra, 89% extra)					

D.4.3 TCP Packets

	Ale cluster		ISISTAN cluster		Bianca cluster	
Image	**Intra**	**Extra**	**Intra**	**Extra**	**Intra**	**Extra**
Image 1	5598	21031	0	16786	0	6496
Image 2	7027	26732	0	22403	0	7620
Image 3	8858	34282	0	30253	0	8780
Image 4	9555	37008	0	33438	0	9363
Image 5	10920	43714	0	39940	0	11514
Subtotal	41958	162767	0	142820	0	43773
Total cluster	204725		142820		43773	
Total packets	391318 (11% intra, 89% extra)					

Bibliography

D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *14th IEEE International Symposium on Parallel and Distributed Processing (IPDPS '00), Cancun, Mexico*, pages 520–528. IEEE Computer Society, Washington, DC, USA, 2000. ISBN 0-76950-574-0.

B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*, 28(5):749–771, 2002. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-8191.

G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001. SAGE Publications. ISSN 1094-3420 (print), 1741-2846 (electronic).

G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling Applications on the Grid: A GridLab Overview. *International Journal of High Performance Computing Applications, Special issue on Grid Computing: Infrastructure and Applications*, 17(4):449–466, 2003. SAGE Publications. ISSN 1094-3420 (print), 1741-2846 (electronic).

G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. V. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005. IEEE Computer Society, Piscataway, NJ, USA. ISSN 0018-9219.

J. M. Alonso, V. Hernández, and G. Moltó. GMarte: Grid Middleware to Abstract Remote Task Execution. *Concurrency and Computation: Practice and Experience*, 18(15):2021–2036, 2006. John Wiley & Sons, Inc., Chichester, UK, UK. ISSN 1532-0626.

D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002. ACM Press, New York, NY, USA. ISSN 0001-0782.

Apache Software Foundation. Jakarta Commons Javaflow. `http://jakarta.apache.org/commons/sandbox/javaflow`, 2006.

R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Mcinnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99), Redondo Beach, CA, USA*, pages 115–124. IEEE Computer Society, Washington, DC, USA, 1999. ISBN 0-76950-287-3.

K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, 1996. ISBN 0-20163-455-4.

M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, and J. Webber. Web Service Grids: An Evolutionary Approach. *Concurrency and Computation: Practice and Experience*, 17(2-4):377–389, 2005. John Wiley & Sons, Inc., Chichester, UK, UK. ISSN 1532-0626.

R. Aversa, B. Di Martino, N. Mazzocca, and S. Venticinque. A Resource Discovery Service for a Mobile Agents Based Grid Infrastructure. In L. T. Yang and Y. Pan, editors, *High Performance Scientific and Engineering Computing: Hardware/Software Support*, pages 189–200. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-40207-580-4.

L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying on the Grid, pages 205–229. Springer, Berlin, Heidelberg, and New York, 2006. ISBN 1-85233-998-5.

M. Baker, R. Buyya, and D. Laforenza. Grids and Grid Technologies for Wide-Area Distributed Computing. *Software Practice & Experience*, 32(15):1437–1466, 2002. John Wiley & Sons, Inc., New York, NY, USA. ISSN 0038-0644.

H. Bal, H. Casanova, J. Dongarra, and S. Matsuoka. Application-level Tools. In (Foster and Kesselman, 2003b), pages 463–489.

R. M. Barbosa and A. Goldman. MobiGrid: Framework for Mobile Agents on Computer Grid Environments. In A. Karmouch, L. Korba, and E. Madeira, editors, *Mobility Aware Technologies and Applications - 1st International Workshop (MATA '04), Florianopolis, Brazil*, volume 3284 of *Lecture Notes in Computer Science*, pages 147–157. Springer, 2004. ISBN 3-54023-423-3. ISSN 0302-9743 (print), 1611-3349 (electronic).

P. Bellavista, A. Corradi, C. Federici, R. Montanari, and D. Tibaldi. Security for Mobile Agents: Issues and Challenges. In I. Mahgoub and M. Ilyas, editors, *Handbook of Mobile Computing*, chapter 39, pages 941–958. CRC Press, 2004. ISBN 0-84931-971-4.

F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel Distributed Systems*, 14(4):369–382, 2003. IEEE Computer Society, Piscataway, NJ, USA. ISSN 1045-9219.

T. Berners-Lee. *Weaving the Web*. Harper, San Francisco, CA, USA, 1999. ISBN 0-06251-586-1.

T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001. Scientific American, Inc. ISSN 0036-8733.

T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Fourth Edition). `http://www.w3.org/TR/REC-xml`, 2006.

CERN. The GridCafé Project. `http://gridcafe.web.cern.ch/gridcafe` (last accessed August 2007), 2007.

C. Chapman, C. Goonatilake, W. Emmerich, M. Farrellee, T. Tannenbaum, M. Livny, M. Calleja, and M. Dove. Condor BirdBath: Web Service Interfaces to Condor. In *2005 UK e-Science All Hands Meeting (AHM '05), Nottingham, UK*, pages 737–744. UK Engineering and Physical Science Research Council, 2005. ISBN 1-90442-553-4.

C. Chapman, P. Wilson, T. Tannenbaum, M. Farrellee, M. Livny, J. Brodholt, and W. Emmerich. Condor Services for the Global Grid: Interoperability Between Condor and OGSA. In *2004 UK e-Science All Hands Meeting (AHM '04), Nottingham, UK*, pages 870–877. UK Engineering and Physical Science Research Council, 2004. ISBN 1-90442-553-4.

M. Chetty and R. Buyya. Weaving Computational Grids: How Analogous Are They with Electrical Grids? *Computing in Science and Engineering*, 4(4):61–71, 2002. IEEE Educational Activities Department, Piscataway, NJ, USA. ISSN 1521-9615.

A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003. Elsevier Science, San Diego, CA, USA. ISSN 0743-7315.

D. C. Chu and M. A. Humphrey. Mobile OGSI.NET: Grid Computing on Mobile Devices. In R. Buyya, editor, *5th International Workshop on Grid Computing (GRID '04), Pittsburgh, PA, USA*, pages 182–191. IEEE Computer Society, Piscataway, NJ, USA, 2004. ISBN 0-7695-2256-4.

L. Chunlin and L. Layuan. Apply Agent to Build Grid Service Management. *Journal of Networks and Computer Applications*, 26(4):323–340, 2003. Academic Press Ltd., London, UK, UK. ISSN 1084-8045.

J. Claessens, B. Preneel, and J. Vandewalle. (How) Can Mobile Agents do Secure Electronic Transactions on Untrusted Hosts? A Survey of the Security Issues and the Current Solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, 2003. ACM Press, New York, NY, USA. ISSN 1533-5399.

W. Codenie, K. D. Hondt, P. Steyaert, and A. Vercammen. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10):71–77, 1997. ACM Press, New York, NY, USA. ISSN 0001-0782.

O. Corcho, P. Alper, I. Kotsiopoulos, P. Missier, S. Bechhofer, and C. Goble. An Overview of S-OGSA: A Reference Semantic Grid Architecture. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):102–115, 2006. Elsevier Science, Amsterdam, The Netherlands. ISSN 1570-8268.

F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003. ACM Press, New York, NY, USA. ISSN 0001-0782.

K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid Information Services for Distributed Resource Sharing. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01), San Francisco, CA, USA*, pages 181–194. IEEE Computer Society, Washington, DC, USA, 2001. ISBN 0-76951-296-8.

J. S. Danaher, I. A. Lee, and C. E. Leiserson. Programming with Exceptions in JCilk. *Science of Computer Programming, Special Issue on Synchronization and Concurrency in Object-Oriented Languages*, 63(2):147–171, 2006. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-6423.

B. V. Dasarathy, editor. *Nearest Neighbor (NN) Norms: Nn Pattern Classification Techniques.* IEEE Computer Society, Los Alamitos, CA, USA, 1991. ISBN 0-8186-8930-7.

T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, 3(1-2):75–90, 2005. Springer. ISSN 1570-7873 (print), 1572-9814 (electronic).

B. Di Martino and O. F. Rana. Grid Performance and Resource Management Using Mobile Agents. In V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 251–263. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-40207-693-2.

Distributed.net. The Distributed.Net Project. `http://www.distributed.net` (last accessed August 2007), 1997.

J. Dongarra and D. Walker. MPI: A Standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996. Stichting Akademisch Rekencentre Amsterdam, Amsterdam, The Netherlands. ISSN 0168-7875.

T. B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Foster City, CA, USA, 1998. ISBN 0-76458-043-4.

R. Englander. *Developing Java Beans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. ISBN 1-56592-289-1.

D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing Among Workstation Clusters. *Future Generation Computer Systems*, 12(1):53–65, 1996. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

M. Feilner. *OpenVPN: Building and Integrating Virtual Private Networks*. Packt Publishing Ltd., Birmingham, UK, 2006. ISBN 1-90481-185-X.

J. F. Ferreira, J. L. Sobral, and A. J. Proenca. JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06), Singapore*, pages 301–304. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-76952-585-7.

Folding@home. The Folding@home Project. `http://folding.stanford.edu` (last accessed August 2007), 2000.

I. Foster. What is the Grid? A Three Point Checklist. *Grid Today*, 1(6), 2002. `http://www.gridtoday.com/02/0722/100136.html`.

I. Foster. The Grid: Computing without Bounds. *Scientific American*, 288(4), 2003. Scientific American, Inc. ISSN 0036-8733.

I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Network and Parallel Computing - IFIP International Conference (NPC '05), Beijing, China*, volume 3779, pages 2–13. Springer, 2005. ISBN 3-54029-810-X. ISSN 0302-9743 (print), 1611-3349 (electronic).

I. Foster and C. Kesselman. Concepts and Architecture. In (Foster and Kesselman, 2003b), pages 37–63.

I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann Publishers Inc., San Francisco, CA, USA, 2003b. ISBN 1-55860-933-4.

I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001. SAGE Publications. ISSN 1094-3420 (print), 1741-2846 (electronic).

I. Foster, C. Kesselman, and S. Tuecke. The Open Grid Services Architecture. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 215–257. John Wiley & Sons Inc., New York, NY, USA, 2003. ISBN 0-47085-319-0.

V. W. Freeh. A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*, 34(1):50–65, 1996. Elsevier Science, Orlando, FL, USA. ISSN 0743-7315.

A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 0098-5589.

M. Fukuda, K. Kashiwagi, and S. Kobayashi. AgentTeamwork: Coordinating Grid-Computing Jobs with Mobile Agents. *Applied Intelligence, Special Issue on Agent-Based Grid Computing*, 25(2):181–198, 2006. Springer, Dordrecht, The Netherlands. ISSN 0924-669X.

D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. On Building Parallel and Grid Applications: Component Technology and Distributed Services. *Cluster Computing*, 8(4):271–277, 2005. Springer, New York, NY, USA. ISSN 1386-7857 (print), 1573-7543 (eletronic).

J. D. Garofalakis, Y. Panagis, and E. S. A. K. Tsakalidis. Contemporary Web Service Discovery Mechanisms. *Journal of Web Engineering*, 5(3):265–290, 2006. Rinton Press, Princeton, NJ, USA. ISSN 1540-9589.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-57108-0.

Globus Alliance. The Java CoG Kit. `http://wiki.cogkit.org/index.php/Java_CoG_Kit` (last accessed August 2007), 2006.

T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications - High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006. Scientific Publishers OWN.

P. Gotthelf, M. Mendoza, A. Zunino, and C. Mateos. GMAC: An Overlay Multicast Network for Mobile Agents. In *6th Argentinian Symposium on Technology (ASAI '05 - 34th JAIIO), Rosario, Santa Fé, Argentina*. Sociedad Argentina de Informática e Investigación Operativa (SADIO), 2005.

J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC '00), Pittsburgh, Pennsylvania, USA*, pages 43–50. IEEE Computer Society, Washington, DC, USA, 2000. ISBN 0-76950-783-2.

GRIDS Laboratory. The GridBus Project. `http://www.gridbus.org` (last accessed August 2007), 2007.

Q. Ho, T. Hung, W. Jie, H. Chan, E. Sindhu, S. Ganesan, T. Zang, and X. Li. GRASG - A Framework for 'Gridifying' and Running Applications on Service-Oriented Grids. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06), Singapore*, pages 305–312. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-76952-585-7.

Q. Ho, Y. Ong, and W. Cai. 'Gridifying' Aerodynamic Design Problem Using GridRPC. In M. Li, X. Sun, Q. Deng, and J. Ni, editors, *Grid and Cooperative Computing - 2nd International Workshop (GCC '03), Shanghai, China*, volume 3032 of *Lecture Notes in Computer Science*, pages 83–90. Springer, 2003. ISBN 3-54021-988-9. ISSN 0302-9743 (print), 1611-3349 (electronic).

E. Huedo, R. S. Montero, and I. M. Llorente. A Framework for Adaptive Execution in Grids. *Software: Practice and Experience*, 34(7):631–651, 2004. John Wiley & Sons Inc., New York, NY, USA. ISSN 0038-0644 (print), 1097-024X (electronic).

M. N. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005. IEEE Computer Society, Piscataway, NJ, USA. ISSN 1089-7801.

J. Hunter and W. Crawford. *Java Servlet Programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998. ISBN 1-56592-391-X.

IBM alphaWorks. Grid Application Framework for Java. `http://alphaworks.ibm.com/tech/GAF4J`, 2004.

H. Imade, R. Morishita, I. Ono, N. Ono, and M. Okamoto. A Grid-Oriented Genetic Algorithm Framework for Bioinformatics. *New Generation Computing*, 22(2):177–186, 2004. Springer Ohmsha, Tokyo, Japan. ISSN 0288-3635.

F. Ishikawa, N. Yoshioka, Y. Tahara, and S. Honiden. Towards Synthesis of Web Services and Mobile Agents. In Z. Maamar, C. Lawrence, D. Martin, B. Benatallah, K. Sycara, and T. Finin, editors, *Workshop on Web Services and Agent-Based Engineering (WSABE '04), 3rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS'2004), New York, NY, USA*. 2004.

W. A. Jansen. Countermeasures for Mobile Agent Security. *Computer Communications*, 23(17):1667–1676, 2000. Elsevier Science, Amsterdam, The Netherlands. ISSN 0140-3664.

R. Johnson. J2EE Development Frameworks. *Computer*, 38(1):107–110, 2005. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 0018-9162.

R. E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, 1997. ACM Press, New York, NY, USA. ISSN 0001-0782.

A. Jugravu and T. Fahringer. JavaSymphony, a Programming Model for the Grid. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, 21(1):239–247, 2005. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

P. Kacsuk and G. Sipos. Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal. *Journal of Grid Computing*, 3(3-4):221–238, 2005. Springer. ISSN 1570-7873 (print), 1572-9814 (electronic).

N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003. Elsevier Science, San Diego, CA, USA. ISSN 0743-7315.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001. ACM Press, New York, NY, USA. ISSN 0001-0782.

G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, New York, NY, USA, 1997.

T. Kielmann, A. Merzky, H. Bal, F. Baude, D. Caromel, and F. Huet. Grid Application Programming Environments. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids - Workshop on Future Generation Grids, Dagstuhl, Germany*, pages 286–306. Springer, 2006. ISBN 0-38727-935-0.

S. D. Kim and S. H. Chang. A Systematic Method to Identify Software Components. In *11th Asia-Pacific Software Engineering Conference (APSEC '04), Busan, Korea*, pages 538–545. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-76952-245-9. ISSN 1530-1362.

P. Z. Kolano. Facilitating the Portability of User Applications in Grid Environments. In J. Stefani, I. M. Demeure, and D. Hagimont, editors, *Distributed Applications and Interoperable Systems - 4th IFIP WG6.1 International Conference (DAIS '03), Paris, France*, volume 2893 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2003. ISBN 3-54020-529-2. ISSN 0302-9743 (print), 1611-3349 (electronic).

J. Kommineni and D. Abramson. GriddLeS Enhancements and Building Virtual Applications for the Grid with Legacy Components. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005, European Grid Conference, Amsterdam, The Netherlands*, volume 3470 of *Lecture Notes in Computer Science*, pages 961–971. Springer, 2005. ISBN 3-54026-918-5. ISSN 0302-9743 (print), 1611-3349 (electronic).

D. Kotz and R. S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, 1999. ACM Press, New York, NY, USA. ISSN 0163-5980.

H. Kreger. Web Services Conceptual Architecture (WSCA 1.0). Technical report, IBM Corporation, 2001.

D. B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, 1999. ACM Press, New York, NY, USA. ISSN 0001-0782.

E. Lee, B. Lee, W. Shin, and C. Wu. A Reengineering Process for Migrating from an Object-Oriented Legacy System to a Component-Based System. In *Design and Assessment of Trustworthy Software-Based Systems - 27th International Conference on Computer Software and Applications (COMPSAC '03), Dallas, Texas, USA*, pages 336–341. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-76952-020-0.

D. Levine and M. Wirt. Interactivity with Scalability: Infrastructure for Multiplayer Games. In (Foster and Kesselman, 2003b), pages 167–178.

L. Loewe. Evolution@home: Observations on Participant Choice, Work Unit Variation and Low-Effort Global Computing. *Software: Practice and Experience*, 37(12):1289–1318, 2007. John Wiley & Sons, Inc., New York, NY, USA. ISSN 0038-0644 (print), 1097-024X (electronic).

P. H. M. Maia, N. C. Mendonça, V. Furtado, W. Cirne, and K. Saikoski. A Process for Separation of Crosscutting Grid Concerns. In *2006 ACM Symposium on Applied Computing, Dijon, France*, pages 1569–1574. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-108-2.

J. Martin. Web Services: The Next Big Thing. *XML Journal*, 2(5), 2001. `http://xml.sys-con.com/read/40192.htm`.

C. Mateos, M. Crasso, A. Zunino, and M. Campo. Adding Semantic Web Services Matching and Discovery Support to the MoviLog Platform. In M. Bramer, editor, *Artificial Intelligence in Theory and Practice - IFIP 19th World Computer Congress (WCC '06), Santiago, Chile*, volume 217 of *IFIP International Federation for Information Processing*, pages 51–60. Springer, Boston, MA, USA, 2006a. ISBN 0-38734-654-6. ISSN 1571-5736.

C. Mateos, M. Crasso, A. Zunino, and M. Campo. Supporting Ontology-Based Semantic Matching of Web Services in MoviLog. In J. S. ao Sichman, editor, *Advances in Artificial Intelligence - 2nd International Joint Conference, 10th Ibero-American Conference on AI, 18th Brazilian AI Symposium (IBERAMIA-SBIA '06), Ribeirão Preto, Brazil*, volume 4140 of *Lecture Notes in Computer Science*, pages 390–399. Springer, 2006b. ISBN 3-54045-462-4.

C. Mateos, A. Zunino, and M. Campo. Integrating Intelligent Mobile Agents with Web Services. *International Journal of Web Services Research*, 2(2):85–103, 2005. Idea Group Inc., Hershey, PA, USA. ISSN 1545-7362 (print), 1546-5004 (electronic).

C. Mateos, A. Zunino, and M. Campo. A Survey on Approaches to Gridification. *Software: Practice and Experience*, 2007a. John Wiley & Sons, Inc., New York, NY, USA. ISSN 0038-0644 (print), 1097-024X (electronic). To appear.

C. Mateos, A. Zunino, and M. Campo. Extending MoviLog for Supporting Web Services. *Computer Languages, Systems & Structures*, 33(1):11–31, 2007b. Elsevier Science. ISSN 1477-8424.

C. Mateos, A. Zunino, and M. Campo. *Modern Technologies in Web Services Research*, chapter Mobile Agents Meet Web Services, pages 98–121. Cybertech Publishing, Hershey, PA, USA, 2007c. ISBN 1-59904-280-0.

C. Mateos, A. Zunino, and M. Campo. JGRIM: An Approach for Easy Gridification of Applications. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, 24(2):99–118, 2008. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

A. Mauthe and O. Heckmann. Distributed Computing - GRID Computing. In R. Steinmetz and K. Wehrle, editors, *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2005. ISBN 3-540-29192-X. ISSN 0302-9743 (print), 1611-3349 (electronic).

R. Montanari, E. Lupu, and C. Stefanelli. Policy-Based Dynamic Reconfiguration of Mobile-Code Applications. *Computer*, 37(7):73–80, 2004. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 0018-9162.

H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical report, GridRPC Working Group, 2005.

A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. The Legion Support for Aadvanced Parameter-Space Studies on a Grid. *Future Generation Computer Systems*, 18(8):1033–1052, 2002. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

M. O. Neary and P. Cappello. Advanced Eager Scheduling for Java-Based Adaptive Parallel Computing. *Concurrency and Computation: Practice and Experience*, 17(7-8):797–819, 2005. John Wiley & Sons Inc., New York, NY, USA. ISSN 1532-0626.

T. Nguyena and P. Kuonen. Programming the Grid with POP-C++. *Future Generation Computer Systems*, 23(1):23–30, 2007. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

OASIS Consortium. UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, `http://uddi.org/pubs/uddi_v3.htm`, 2004.

OASIS Consortium. Web Services Resource Framework (WSRF) - Primer v1.2. Committee Draft 02. `http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf`, 2006.

OGSA-WG. Defining the Grid: A Roadmap for OGSA Standards. `http://www.ogf.org/documents/GFD.53.pdf`, 2005.

A. L. Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Boston, MA, USA, 1998. ISBN 0-20163-386-8.

L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Volunteer Computing Systems Using Java. *Future Generation Computer Systems, Special Issue on Metacomputing*, 15(5-6):675–686, 1999. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

C. Spyrou, G. Samaras, E. Pitoura, and P. Eviripidou. Mobile Agents for Wireless Computing: The Convergence of Wireless Computational Models with Mobile-Agent Technologies. *Mobile Networks and Applications*, 9(5):517–528, 2004. Kluwer Academic Publishers, Hingham, MA, USA. ISSN 1383-469X.

H. Stockinger. Defining the Grid: A Snapshot on the Current View. *Journal of Supercomputing*, 42(1):3–17, 2007. Springer, Dordrecht, The Netherlands. ISSN 0920-8542.

Sun Microsystems. Sun N1 Grid Engine 6. `http://www.sun.com/software/gridware` (last accessed August 2007), 2005.

A. Suna, G. Klein, and A. E. Fallah-Seghrouchni. Using Mobile Agents for Resource Sharing. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '04), Beijing, China*, pages 389–392. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-76952-101-0.

G. Taffoni, D. Maino, C. Vuerli, G. Castelli, R. Smareglia, A. Zacchei, and F. Pasian. Enabling Grid Technologies for Planck Space Mission. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, 23(2):189–200, 2007. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

H. Takemiya, K. Shudo, Y. Tanaka, and S. Sekiguchi. Constructing Grid Applications Using Standard Grid Middleware. *Journal of Grid Computing*, 1(2):117–131, 2003. Springer. ISSN 1570-7873 (print), 1572-9814 (electronic).

I. J. Taylor. *From P2P to Web Services and Grids: Peers in a Client/Server World*. Computer Communications and Networks. Springer, 2005. ISBN 1-85233-869-5.

D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335. John Wiley & Sons Inc., New York, NY, USA, 2003. ISBN 0-47085-319-0.

A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora, and M. Pathak. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140, 2002. Elsevier Science, New York, NY, USA. ISSN 0164-1212.

D. Tschumperlé and R. Deriche. Vector-Valued Image Regularization with PDE's: A Common Framework for Different Applications. In *2003 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '03), Madison, WI, USA*, volume 1, pages 651–656. IEEE Computer Society, Los Alamitos, CA, USA, 2003. ISBN 0-7695-1900-8.

S. Vadhiyar and J. Dongarra. Self Adaptability in Grid Computing. *Concurrency and Computation: Practice and Experience, Special Issue on Grid Performance*, 17(2-4):235–257, 2005. John Wiley & Sons Inc., New York, NY, USA. ISSN 1532-0626.

R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and Efficient Java-Based Grid Programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, 2005a. Warsaw School of Social Psychology. ISSN 1895-1767.

R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: A Flexible and Efficient Java Based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005b. John Wiley & Sons Inc., New York, NY, USA. ISSN 1532-0626.

L. Vasiu and Q. H. Mahmoud. Mobile Agents in Wireless Devices. *Computer*, 37(2):104–105, 2004. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 0018-9162.

S. J. Vaughan-Nichols. Web Services: Beyond the Hype. *Computer*, 35(2):18–21, 2002. IEEE Computer Society, Los Alamitos, CA, USA. ISSN 0018-9162.

S. Venugopal, R. Buyya, and K. Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys*, 38(1):1–53, 2006. ACM Press, New York, NY, USA. ISSN 0360-0300.

W3C Consortium. SOAP Version 1.2 Part 0: Primer (Second Edition). W3C Recommendation, `http://www.w3.org/TR/soap12-part0`, 2007a.

W3C Consortium. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Candidate Recommendation, `http://www.w3.org/TR/wsdl20`, 2007b.

C. Walls and R. Breidenbach. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 2005. ISBN 1-93239-435-4.

B. Wang, Z. Xu, C. Xu, Y. Yin, W. Ding, and H. Yu. A Study of Gridifying Scientific Computing Legacy Codes. In H. Jin, Y. Pan, N. Xiao, and J. Sun, editors, *Grid and Cooperative Computing - 3rd International Conference (GCC '04), Wuhan, China*, volume 3251 of *Lecture Notes in Computer Science*, pages 404–412. Springer, 2004. ISBN 3-54023-564-7. ISSN 0302-9743 (print), 1611-3349 (electronic).

D. Webb and A. L. Wendelborn. The PAGIS Grid Application Environment. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Computational Science - ICCS 2003, Melbourne, Australia and St. Petersburg,*

Russia, volume 2659 of *Lecture Notes in Computer Science*. Springer, 2003. ISBN 3-54040-196-2. ISSN 0302-9743 (print), 1611-3349 (electronic).

R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999. Elsevier Science, Amsterdam, The Netherlands. ISSN 0167-739X.

G. Wrzesinska, J. Maassen, K. Verstoep, and H. E. Bal. Satin++: Divide-and-Share on the Grid. In *2nd IEEE International Conference on e-Science and Grid Computing (E-SCIENCE '06), Amsterdam, Netherlands*, page 61. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-76952-734-5.

C. S. Yeo, R. Buyya, M. D. de Assunção, J. Yu, A. Sulistio, S. Venugopal, and M. Placek. Utility Computing on Global Grids. In H. Bidgoli, editor, *The Handbook of Computer Networks*, chapter 143. John Wiley & Sons Inc., New York, NY, USA, 2007. ISBN 0-47178-461-3.

A. Zunino. *Movilidad Reactiva por Fallas: Un Modelo de Movilidad para Simplificar el Desarrollo de Agentes Móviles*. PhD Dissertation, Universidad del Centro de la Provincia de Buenos Aires, Campus Universitario - Paraje Arroyo Seco, Tandil, Buenos Aires, Argentina, 2003. (In Spanish).

A. Zunino, C. Mateos, and M. Campo. Enhancing Agent Mobility Through Resource Access Policies and Mobility Policies. In *5th Encontro Nacional de Inteligência Artificial (ENIA '05), 15th Congresso da Sociedade Brasileira de Computação (SBC '05), Saõ Leopoldo, RS, Brasil*, pages 851–860. 2005a.

A. Zunino, C. Mateos, and M. Campo. Reactive Mobility by Failure: When Fail Means Move. *Information Systems Frontiers, Special Issue on Mobile Computing and Communications*, 7(2):141–154, 2005b. Springer, Dordrecht, The Netherlands. ISSN 1387-3326.