* Manuscript
This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

JGRIM: An approach for easy gridification of applications

Cristian Mateos *,[1] Alejandro Zunino [1] Marcelo Campo [1]

ISISTAN Research Institute - UNICEN. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 43-9682. Fax.: +54 (2293) 43-9681

Abstract

The term "Grid" was coined to describe a form of distributed computing in which hardware and software resources from disparate sites are virtualized to provide applications with a single, powerful computing infrastructure. In fact, the term comes from an analogy with the electrical power grid infrastructure, since researchers expect applications to access Grid resources as transparently as electricity is now consumed. However, "plugging" applications to the Grid is still very difficult because current toolkits force programmers to take into account many complex details when coding applications for the Grid. This paper presents JGRIM, a middleware for easy "gridification" of ordinary Java applications. The main goal of JGRIM is to make the task of porting applications to the Grid easier and, at the same time, provide facilities to easily address efficiency issues when gridifying an application. An experimental evaluation showing the feasibility of our approach and some of its benefits is also reported.

Key words: Grid Computing, Gridification, Service-Oriented Grids, Mobile agents, Dependency Injection

1 Introduction

Grid Computing [1] is a form of distributed computing in which resources such as processing power, disk storage, applications and data, often spread across different physical locations and administrative domains, are shared and optimized through virtualization and collective management. The main concept in Grid Computing is

* Corresponding author.
Email addresses: cmateos@exa.unicen.edu.ar (Cristian Mateos), azunino@exa.unicen.edu.ar (Alejandro Zunino).
[1] Also CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

the Grid [2] [2], a distributed computing infrastructure whose objective is to provide safe and coordinated computational resource sharing between organizations.

The first attempts to provide Grid-like infrastructures were focused on linking supercomputing nodes to support compute-intensive, large-scale scientific problems and applications [3]. During the first half of 1990s, the inception and increasing popularity of Internet standards enabled the materialization of massively distributed infrastructures for resource sharing. An early example is Volunteer Computing [4]: users from all over the world donate CPU cycles by running programs that download and analyze scientific data while their PCs are idle. Then, the first middlewares for implementing Grid applications appeared. Examples of popular Grid middlewares are Legion [5], Condor [6,7] and Globus [8], this latter baptized by Ian Foster as the "Linux of the Grid".

Since the notion of "Grid" was introduced, many technological changes have occurred in both hardware and software, and one of the most important ones is the wide acceptance of Web services [9]. Web Services technologies such as SOAP [10], WSDL [11] and UDDI [12] 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, major Grid standardization efforts such as the Open Grid Services Architecture (OGSA) [13] and the Web Services Resource Framework (WSRF) [14] 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 [15]. For instance, research has been being done to integrate Condor with Web Services [16,17], and Globus has recently embraced WSRF.

There is a widely-known analogy between the computational Grid and the electrical power grid, from where the former takes its name. The Grid promises to let users access computational resources as easily and transparently as electrical power can be consumed by appliances from a wall socket [18,19]. Indeed, one of the goals of Grid computing is to allow software developers to code the logic of some application, deploy it on the Grid, and let the Grid to locate and use 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. Unfortunately, the analogy does not completely hold yet since it is too hard to "gridify" an application without explicitly rewriting it to make it Grid-aware.

We believe that the cause of this problem is that many of the current toolkits for Grid programming provide 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 resources and services, thus making

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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

maintainability, legibility, testing and portability to different Grid platforms very hard. Furthermore, integrating existing source code onto the Grid requires in most cases to rewrite significant portions of the application to use those APIs. These problems are addressed by toolkits 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 codes with a software entity that isolates the complex details of the underlying Grid [20–22]. 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 (e.g. distribute 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.

To cope with the problems mentioned above, we propose JGRIM, a middleware for easy "gridification" of applications. In essence, JGRIM aims at dealing with the above tradeoff 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. Basically, 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. JGRIM imposes little code modifications. In addition, its programming model shares many similarities with widely used models for Java programming such as JavaBeans. JGRIM promotes separation of concerns between application logic and Grid behavior. Therefore, we conceive *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 to it.

JGRIM is materialized as an implementation of GRIM (Generalized Reactive Intelligent Mobility), a generic agent programming and execution model which allows developers to easily manage mobility of agents and resources. Grid applications in JGRIM (i.e. a gridified application) are application-level mobile services called MGS (Mobile Grid Services) which interact with other MGSs and use middleware-level Grid services for mobility, execution, resource brokering and monitoring. 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.

The main contribution of this paper lies in offering a solution to the gridification problem inherent to service-oriented Grids. In general, current middlewares and toolkits demand the developer to know in advance many platform-related details before an application can take advantage of Grid services. On the contrary, JGRIM lets developers to easily code and deploy Grid applications while keeping them away from most Grid-related details. Besides providing mechanisms for easily porting applications to the Grid, our proposal aims at addressing performance

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

issues through a mobile agent-based, easy-to-tune application execution support.

The remainder of this article is organized as follows. The next section surveys the most relevant related work. Section 3 presents JGRIM. Then, section 4 introduces the GRIM model. Later, section 5 reports a comparison between JGRIM, Proactive and Ibis, two Java-based platforms for implementing Grid applications. Finally, in section 6 concluding remarks and future work are presented.

2 Related work

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. Some of the most relevant examples are Proactive [23], Ibis [24], JavaSymphony [25], XCAT [26] and Gridbus [27].

Proactive is a Java-based middleware for object-oriented parallel, mobile and distributed computing. A typical Proactive application is composed of a number of mobile entities called *active objects*. Active objects serve methods calls originated from other objects, and invoke methods implemented by other local or remote objects. Method calls are asynchronously handled based on the *wait-by-necessity* mechanism, which transparently blocks the requester upon the first attempt to access the results of any previously issued call. The weak point of the approach is that active object creation, lookup and mobility are in charge of the programmer. Consequently, the functionality for managing parallelism and object migration is usually mixed with the application logic, which renders maintainability and portability significantly difficult.

Ibis is an middleware whose goal is to provide an efficient Java-based platform for Grid programming. Ibis consists of a communication library, and a variety of programming models, mostly for developing applications as a number of components exchanging messages through RMI or an MPI-like communication protocol. Unfortunately, Ibis does not offer support for using ubiquitous Grid technologies like SOAP, WSDL and UDDI. An interesting subsystem of Ibis is Satin [28,29], which lets programmers to easily parallelize and distributively execute applications based on the divide and conquer paradigm. However, Satin aims to achieve high efficiency only for CPU-intensive applications.

JavaSymphony is a high-level programming model for simplifying the development of performance-oriented, object-based Grid applications. JavaSymphony provides a semi-automatic execution model which deals with migration, parallelism and load balancing of applications, and at the same time allows the programmer to explicitly control such features as needed. However, the source code of applications that use JavaSymphony tends to get mixed up with code for handling non-

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

functional issues such as object migration, threading and method asynchrony. Like Ibis, JavaSymphony offers limited support for using common Grid-related protocols and technologies. Finally, the applicability of JavaSymphony to heterogeneous Grid settings in which hosts can frequently join or leave a computation (e.g. a WAN or the Internet) is still under evaluation.

XCAT is a framework that supports distributed execution of component-based applications. XCAT applications are stateful Grid services composed of one or more *components*. XCAT runs on top of existing Grid middleware, such as Globus, linking application components to concrete Grid services. XCAT also allows existing compiled applications to be gridified using the concept of *application manager*. Basically, each binary is wrapped with a generic component (application manager) that is responsible for managing and monitoring the execution of the application. Applications managers can be connected to each other, and have one special port by which standard components (i.e. those used to represent Grid services) can control them. Though the mapping between application components and Grid services can be established with little coding effort, the user still has to programmatically manage component creation and linking at the application level. Furthermore, opportunities for application tuning largely depend on the facilities the underlying Grid middleware being used offers, as XCAT does not provide easy-to-use support for customizing the way application components interact between each other.

Gridbus is a Java-based resource broker that hides the complexity of the Grid by mapping application execution requests coming from users into a set of jobs whose execution is scheduled and managed on appropriate nodes. Gridbus emphasis is on providing efficient brokering and execution services on data-oriented Grids, that is, those offering infrastructure to manage large amounts of data. To this end, Gridbus extends Nimrod-G [30], a Grid resource brokering model specialized in parameter sweep applications. Gridbus allows legacy applications to run unmodified on a Grid setting and transparently take benefit from Grid services such as resource discovery, scheduling and data transfer. Our work can be viewed as a complement to Gribus because it let developers to leverage existing Grid services from within conventional applications and, at the same time, to easily control the various runtime aspects (e.g. distribution, parallelism, mobility, etc.) of these applications when running on a Grid.

In addition, a number of middlewares for job submission and scheduling can be found in the literature. Platforms such as Condor, Javelin 3.0 [31] and GridWay [32] provide an execution environment for deploying CPU-intensive applications, which are moved between computing nodes to leverage unused processing power. Sadly, these platforms share two main problems. On one hand, they do not provide efficient execution support for non CPU-bound applications (e.g. data-intensive) where other Grid resources such as bandwidth, storage and information services are intensively consumed as well. On the other hand, these middlewares require large amounts of Grid-specific configuration information in order to run an application.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

Examples are the expected execution time, memory usage and number of nodes to use. In addition, Javelin 3.0 and GridWay require applications to be explicitly modified and restructured to exploit job checkpointing and splitting. These facts, in turn, clearly hinder the adoption of these technologies by developers.

Finally, many Java-based toolkits for using Grid services from within ordinary applications have been proposed. For example, the Java CoG Kit [33] provides an interface to Globus-specific services, such as GRAM (execution), GridFTP (data transfer), GSI (security) or MDS (service discovery). The Grid Application Toolkit (GAT) [34] is similar to the Java CoG Kit but it offers a higher level API for accessing services which is independent of the underlying Grid middleware. Lastly, the Grid Application Framework (GAF) [35] is a framework where threads within applications are replaced by task objects that are transparently sent to remote hosts for execution. GAF also provides high-level interfaces to Grid services for resource management, monitoring, data access, and so on. All in all, these toolkits not only force Grid application developers to learn yet another programming API, but also require to modify applications in order to adapt them to the API or framework being used.

To address the above problems, we propose JGRIM, a Java-based middleware for easy development and deployment of Grid applications. JGRIM aims at creating a flexible and transparent gridification layer for users wanting to port applications to the Grid by (a) allowing programmers to access Grid services and resources from applications minimizing the requirement of modifying their source code, (b) offering a simple, service-oriented, mobility-based execution model called GRIM, and (c) providing efficient execution support for Grid applications, and easy-to-use application and middleware-level tuning mechanisms.

JGRIM aims at preserving the integrity of the application logic. To this end, JGRIM lets developers to focus first on coding the pure functional behavior of their applications, and *then* to non-intrusively add Grid behavior to them, that is, without using any Grid API. Even when gridifying with JGRIM may require users to modify their applications, modifications are minimal. Furthermore, the JGRIM API only needs to be explicitly accessed if users want to perform domain-specific application tuning and, in those cases, the necessary subset of the API is very small and easy to understand. Finally, the programming model of JGRIM is essentially based on a component-based paradigm. Conceptually, programming with JGRIM does not differ too much from using JavaBeans, a very popular programming model among Java developers.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

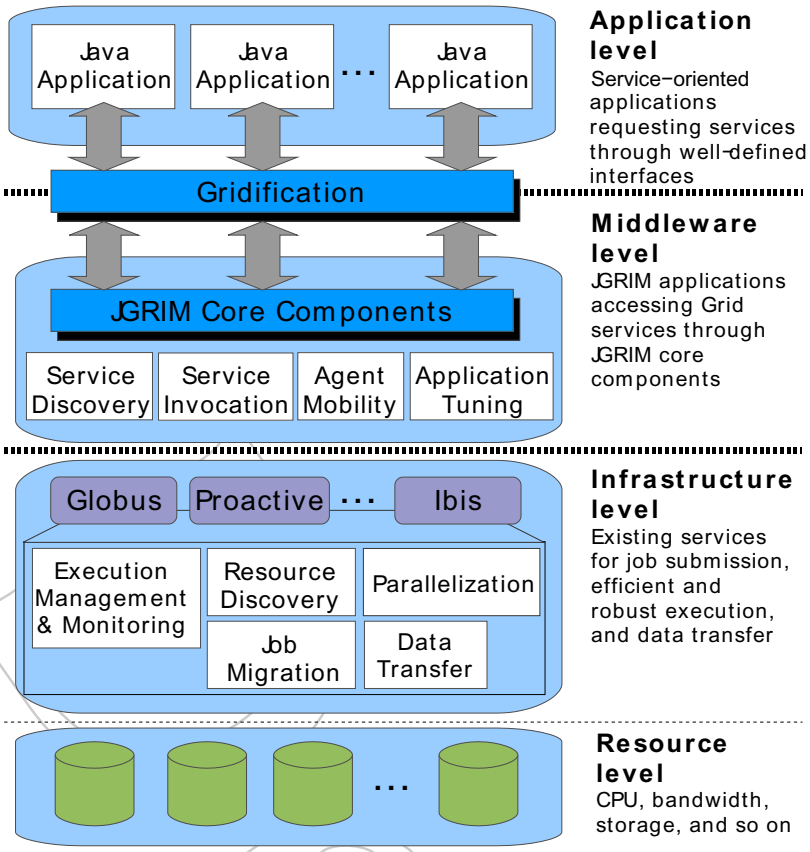


Fig. 1. The JGRIM approach: a layered view

3 JGRIM

JGRIM is a Java-based middleware for creating and deploying applications on service-oriented Grids. The utmost goal of JGRIM is to make materialization and deployment of Grid applications 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 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 invoking these services from within the application logic.

It is important to note that our approach does not aim at providing yet another infrastructure for dealing with resource management on the Grid. Instead, our vision is to create a layer whereby Java applications are *effortlessly* transformed to applications which are furnished with specialized "components" to take advantage of Grid services. These components act as a glue between applications and the Grid, using its metadata services for achieving service discovery, invocation and efficient interaction with services, thus leveraging and enhancing the underlying services provided by existing Grid platforms.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

Figure 1 depicts an overview of JGRIM. As shown, JGRIM sits on top of existing Grid infrastructures by adding a middleware layer that enable component-based Java applications to seamlessly use Grid services. Service requests originated at the application level are handled by two middleware-level components named `Service Discovery` and `Service Invocation`, which are in charge of dealing with service discovery and interaction within the Grid, respectively. JGRIM also provides components for transparently adapting the execution of gridified applications to the characteristics of the Grid being used, thus improving application performance. In this paper we are mostly concerned with achieving easy gridification by mapping Java applications to efficient JGRIM applications and, to a lesser extent, with the mechanisms for bridging JGRIM components with infrastructure-level services.

The next subsection briefly introduces the concepts on which our approach is based: Grid services and mobile agents. Then, subsection 3.2 provides details on the internals of the applications once they have been gridified with JGRIM. Subsections 3.4 and 3.5 describe the gridification process in detail.

3.1 Grid services and mobile agents

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 [36]. 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 (reaction, perception, deliberation, autonomy, etc.). Some of the most significant advantages of mobile agents are their support for disconnected operations, heterogeneous systems integration, robustness and scalability [37].

A mobile agent has the capacity to migrate to the location where a resource is hosted, thus allowing local 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 well suited to be managed by mobile agents [38]. Scheduling, brokering, monitoring and coordination are inherently high-level tasks that require agents' abilities such as autonomy, proactivity, mo-

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

bility and negotiation. Indeed, in the context of massively distributed Grids, migratory capabilities will be very important. 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 [39], 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 [37].

Using mobility within the Grid is not a new idea. For example, mobile agents have been successfully employed for job submission and management [40,41], resource sharing [42], and resource discovery [43,44]. In addition, Grid infrastructures such as Cactus [45], Condor [7], GridWay [32] and GrADS [46] also rely on mobility for both application scheduling and execution. However, their migration frameworks use traditional process migration techniques, while our work 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.

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. As we will see in the next subsections, this process is semi-automatic, as it requires: (a) modification of source code in order to obey simple and standard object-oriented code conventions, (b) user information about the interface of external Grid services accessed by the application, and (c) assembling and deployment of the outputs of (a) and (b), which are performed automatically by JGRIM.

Applications after passing through the JGRIM gridification process become Web Services with migration capabilities called MGS (Mobile Grid Service). Basically, an MGS 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 (Java class), which acts as a bridge between the WSDL and the mobile part of the service. WSDL [11] is a well-known XML-based language for describing Web 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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

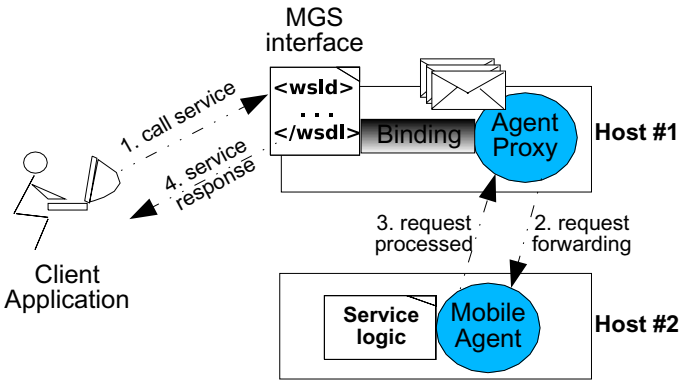


Fig. 2. Components of an MGS

MGS is not supplied by the developer, since it is automatically built from the public methods of the MGS implementation (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 subsection presents further details on how this is done.

When an MGS is sent to a host for deploying, its WSDL 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 service request, in a piggybacking fashion. Moreover, each proxy maintains a "mail box" 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 2 depicts the anatomy of a gridified application.

The run-time support for MGSs is implemented through Java servlets [47]. It provides low-level services to agents such as execution, mobility, 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 [48], a library from Apache for capturing the execution state of a running Java thread.

3.3 Dependency Injection

Central to JGRIM is the concept of Dependency Injection (DI) [49], a form of the Inversion of Control notion found in object-oriented frameworks. The idea behind

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

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 run-time platform in charge of binding clients to service providers.

In the following paragraphs we will briefly illustrate the concept of DI through an example. Let us suppose we have a component 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 GridFTP [3] for transferring the file. The code of this component is very simple: setup a GridFTP connection to the remote site, transfer and parse the file, and then iterate the results locally in order to display book information. The code implementing 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);
    FileRandomIO sink = new FileRandomIO(new RandomAccessFile("local-books.info"));
    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. But there is more: depending on the way information is accessed, a different set of configuration parameters could be required (e.g. passwords, endpoints, URLs, etc.). In such a case, `BookLister` must also be modified to include the necessary constructors/setters methods.

[3] GridFTP is an FTP-based, high-performance, secure, reliable data transfer protocol for Grid environments which is distributed with the Globus Toolkit.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

The DI version of the listing component could include an interface (`BookSource`) by which `BookLister` accesses the book information, and 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, 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, filter books by [topic], and return results
     */
  }
  ...
}

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
  }
}
```

Now, we have to 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 XML file, which specifies concrete implementations and configuration information for all the components of an application along with the dependencies that exist between them. In the rest of the paper, the examples use Spring [50,49] as the DI container. The configuration file for the example is:

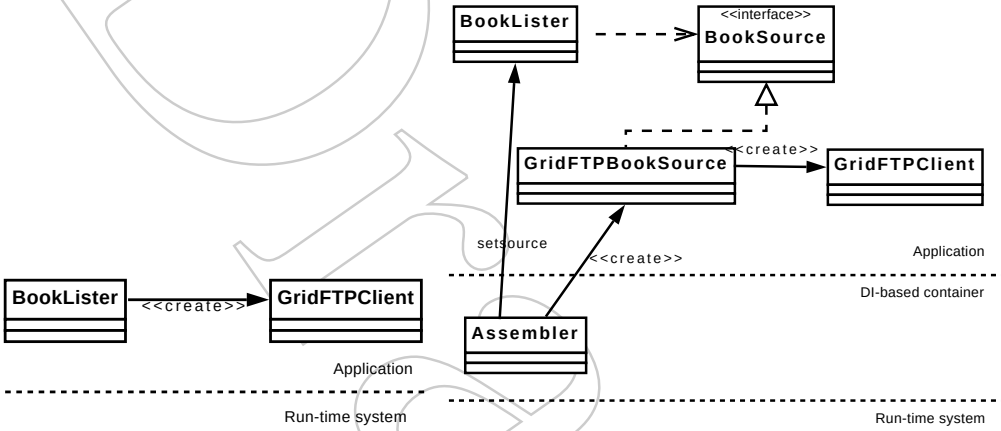
```
<?xml version="1.0" encoding="UTF-8" ?>
<components>
  <component id="myLister" class="BookLister">
    <dependency name="Source">mySource</dependency>
  </component>
```

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

```
  <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 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 extensibility. On the contrary, in the DI version shown on the right the code for dealing with creating and configuring GridFTP is partially replaced by configuration information placed on a separate file, which is processed at run-time by an assembling element supplied by the DI container.



(a) Without using DI (b) Using DI

Fig. 3. Class diagrams for the book listing application

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 [49]. An even more interesting implication of DI to our work is that dependencies can be completely managed by a middleware to add Grid behavior to ordinary Java applications. Specifically, Grid functionality such as service discovery, service invocation and parallelization are provided by JGRIM by means of injection of built-in 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.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

3.4 Gridifying applications with JGRIM

Figure 4 summarizes the concepts upon JGRIM is based. User applications are composed of *logic* and *dependencies*. The logic is the set of classes implementing 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. A dependency specifies, by means of a Java interface, the conventions needed to appropriately interact with resources and services. Furthermore, dependencies may have attached a *policy*, which customizes the way those resources and services are accessed. Finally, at the middleware level, both policies and dependencies are materialized as JGRIM built-in components. Basically, these components are intended to provide:

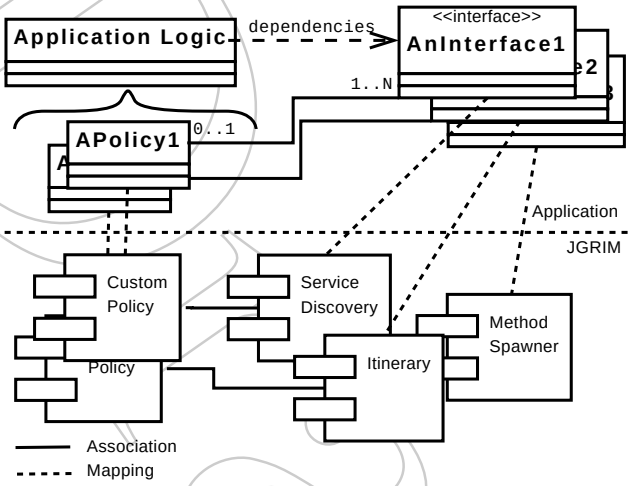


Fig. 4. Dependencies, policies and JGRIM components

- *Service discovery*: By default, all dependencies for an application are assumed to refer to external Web Services, MGSs or any other kind of service-like entity described through a WSDL document. The Java interface (i.e. `BookSource` in the above example) for every dependency of a JGRIM application is in fact transparently associated to a service discovery 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, and invoke operations on that service [4] . In this way, the user is freed not only from the burden of searching services, but also from dealing with low-level details such as addresses, protocols, ports, etc. for invocating services.

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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

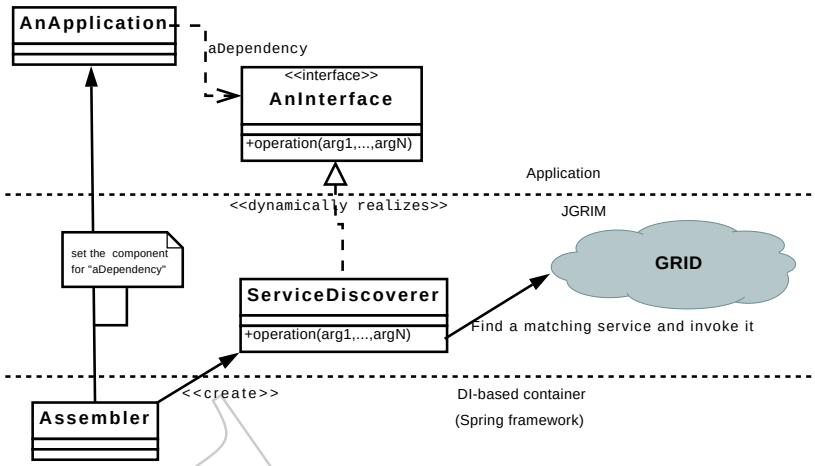


Fig. 5. Dependencies and service discovery components in JGRIM

Figure 5 shows a diagram illustrating how JGRIM isolates applications from discovering and invoking Grid services. The `ServiceDiscoverer` class materializes the service discovery component previously mentioned. Instances of this class, which are automatically injected into the application, act as proxies to Grid services by converting any incoming method call to the corresponding WSDL operation request.

- *Itineraries*: JGRIM supports the concept 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 component which, upon creation, locates the services adhering the interface of the dependency to which the component is associated, and store them into a list. The 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 when 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 a dependency to an itinerary component (`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;
  }
```

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

```
  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 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 component can be viewed as a special type of service discovery 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 components contact all those services which has been initially discovered when the component was injected.

- *Parallelization*: An important aspect of the Grid is parallelization. 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 do 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 let programmers to declare a special interface whose methods corresponds to those operations within a class which should be spawned.

  From our programming model point of view, an interface similar to that of Ibis is considered as a dependency whose methods 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. In this sense, we exploit this notion to transparently inject parallelization: developers can specify "self-dependencies" that are automatically mapped to a *spawner* component, which transparently parallelize the invocation of any method declared within those dependencies. The next section exemplifies the usage of this component.

  At a lower level, this mechanism is complemented by instrumenting the application bytecode so as to wrap the result of a 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. The execution of the method is actually handled parallelly by the spawner in a separate thread, which is also in charge of calling the corresponding `setValue` method on the wrapper object once the results are available.

Another aspect that is represented by means of components is *policy management* (see section 4.2.3). Basically, a policy is a mechanism provided by JGRIM that

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

let programmers to customize the way applications interact with Grid services in order to achieve better performance. Applications use policies mainly for managing MGS mobility. For example, one could instruct an MGS to move to the provider host every time a certain service *S* is executed.

JGRIM maps all policies declarations to components, thus service discovery components can reference policy components. In addition, dependencies can be configured to use policies in order to establish, for example, a custom ordering for accessing services when using itineraries. Conceptually, these policies are known as *agent-level policies*.

Upon gridification, the application logic (i.e. the `AnApplication` in Figure 4) is mapped to an MGS. This is done by automatically modifying the application in order to inherit from the `JGRIMAgent` class. Basically, this class provides 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 components, and combined into a single file which is then "wired" to the MGS behavior. Conceptually, those interfaces are known as *protocols* [5] .

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 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.method()). 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 execution migration.

Wiring of agent behavior and Grid-related behavior to applications is achieved through Spring [50]. Spring is a DI container providing wrappers that make it easy to use many different Java-related technologies and frameworks. Support for DI in Spring is based on JavaBeans, a specification from Sun that defines conventions for writing loosely-coupled reusable software components in Java. Within a Spring application, all of the wiring is configured by means of XML files.

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

[5] From now on, the term should be understood in the context of object-oriented programming

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

3.5 A JGRIM nearest neighbor classifier service

In this section we show the gridification process for the k-nearest neighbor algorithm (k-NN) [51]. k-NN is a 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 `KNN` class declaring three operations (`classifyInstance`, `classifyInstances` and `sameClass`) and a number of helper classes. The idea is to take this 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 be exposed as an MGS (in this case, the KNN class), and a list of pairs *[identifier, JavaInterface]* specifying all the existing dependencies. 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. 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`) identified as *dataset*. The gridified version of the code is:

```
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;
  }
  public double classifyInstance(Instance instance) {...}
  public double[] classifyInstances(Instance[] instances) {...}
  public boolean sameClass(Instance instanceA, Instance instanceB) {...}
```

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

```
}
```

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 component -->
  <bean id="jgrim_dataset" class="jgrim.core.ServiceDiscoverer">
    <property name="expectedInterface">
      <value>DatasetInterface</value>
    </property>
  </bean>
</beans>
```

The MGS 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) `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 qualitative report on the gridification effort for the k-NN algorithm, from the programmer's point of view, can be found in section 5.

3.5.1 Parallelization

A natural way to implement the `sameClass` operation is by issuing two different calls to `classifyInstance`, and then simply compare the results. 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 *spawner*, 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:

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

```
<?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 *spawner* dependency, the programmer must replace the two calls to "this" when using `classifyInstances` within the `sameClass` operation by calls to a fictitious method `getspawn`. 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 6.

3.5.2 Policy management

To illustrate the run-time behavior of our classifier service, let us suppose a scenario consisting of several JGRIM-enabled sites where some of them have a copy of the dataset stored 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 bandwidth 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, both bandwidth and availability of CPU should be taken into account. Accessing a service from a site to which the MGS current location experience bad bandwidth might increase latency. Moreover, processing a block of data on a loaded site might increase response time. We could overcome this situation by attaching a policy to the *dataset* dependency. To code a policy, the programmer must specify his/her access decisions by implementing four separate methods of the `PolicyAdapter` class:

```
import jgrim.core.Constants;
import jgrim.core.PolicyAdapter;
```

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

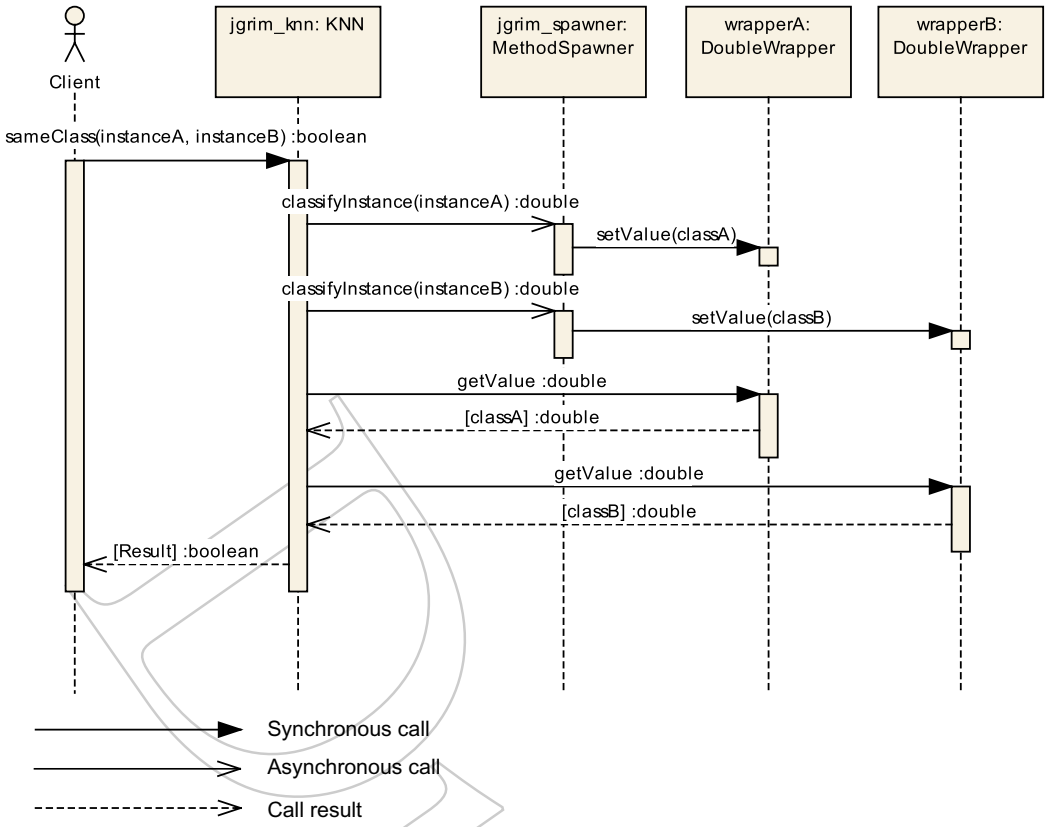


Fig. 6. Parallelization of the `sameClass` operation: sequence diagram

```
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 trA = pr.profile("bandwidth", "localhost", siteA);
    double trB = pr.profile("bandwidth", "localhost", siteB);
    return (trA < trB)? siteA : siteB;
  }
  public void beforeCallPolicy(){
    String site = Profiler.getInstance().leastLoadedSite();
    ((JGRIMAgent)getOwner()).move(site);
  }
  public void afterCallPolicy(){...}
}
```

For simplicity, exception handling is omitted. The previous code works as follows: every time the MGS calls the dataset service, `DatasetPolicy` is evaluated. Firstly, the method `beforeCallPolicy` 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 best bandwidth. After accessing the service, the code placed within `afterCallPolicy` method is executed. For now, it should be clear that it is very easy to tune JGRIM applications by means of policies, which

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

customize the way Grid services are accessed without altering the MGS behavior. In the next section, a more deep discussion on the policy mechanism is presented.

It is worth noting that `DatasetPolicy` will be activated upon invocation on *any* operation from the `DatasetInterface` interface. 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 bandwidth 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. This is a problem suffered by many of the tools proposed for the Grid, as we have pointed out at the beginning of this paper. However, our proposal 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 service mobility and invocation can be delegated to the platform, at the cost potential of decreased performance.

In the next section we focus on describing the conceptual execution model of JGRIM.

4 GRIM

GRIM (Generalized Reactive Intelligent Mobility) is a generic agent execution model based on Reactive Mobility by Failure (RMF) [52]. RMF is a transparent migration mechanism that aims at reducing the effort of developing mobile agents by automating some decisions about mobility. A *failure* is defined as the impossibility of an executing mobile agent to find some needed resource at the current site. Essentially, a failure is handled by RMF by moving the failing mobile agent to a remote site containing instances of the requested resource.

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. GRIM agents are mobile agents composed of *behavior* and *protocols*. The execution environment for agents residing at a physical site is called a *host*, which is introduced in the next subsection. We will use the terms “agent” and “mobile agent” interchangeably throughout the rest of the paper.

Basically, an agent behavior corresponds to the logic implementing that agent. This

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

logic usually makes use of resources external to the agent which are described by means of protocols. A protocol is the interface – methods and conventions – exposed by a resource to which the agent agrees in order to access the resource. Indirectly, protocols indicates 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). It is up to the middleware to bind each operation issued on the file resource with the code providing the actual implementation. For instance, in the Grid context this might be a Web Service providing operations for accessing files or a Grid file system.

GRIM states that, upon an access to 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. Indeed, not all attempts to access external resources trigger traps, but only those points of an agent 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.

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.

4.1 GRIM run-time support

GRIM is an execution model 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, 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 PNS (Protocol Name Servers) agents.

The run-time platform residing at each physical site capable of hosting and providing support for executing GRIM agents is called a *host*. A set of hosts such as all of them know one another conform a *logical network*. A logical network groups hosts belonging to the same application or some closely related applications. In addition, hosts can provide resources such as databases, libraries or services to agents.

Each host contains one or more PNS agents. PNS agents are responsible for man-

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

aging information about protocols offered at each host, and for searching for the list of resource instances matching a given protocol whenever a trap occurs. A host offering resources registers with its local PNSs the protocols associated with these resources. Then, PNS agents announce the new protocols to its peers in the logical network. Figure 7 shows the relationships between agents, a host and its associated PNSs.

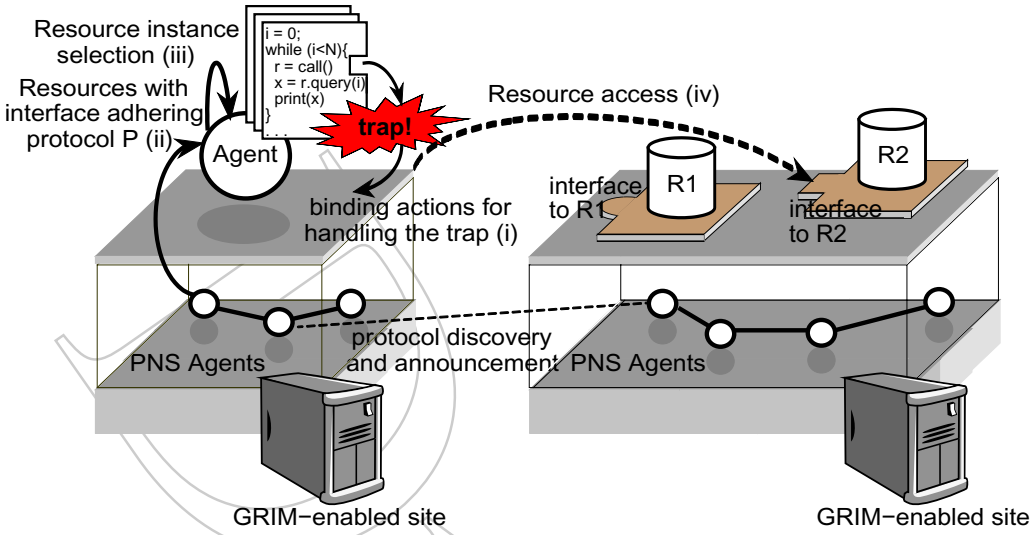


Fig. 7. Overview of GRIM

It is worth pointing out that GRIM does not prescribe any particular mechanism for dealing with protocol information at the PNSs 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 [53], a GRIM-compliant platform for Web development, combines a multicast-based communication facility named GMAC [54] and UDDI registries [12] for managing protocol information. As for JGRIM, protocols are managed similarly to WS-Log.

4.2 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, or the same file could be hosted at more than one site. 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 met-

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

rics 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 database should be done by moving an agent where the database is located, rather than performing a time and bandwidth-consuming copy operation of the data from a remote site. Besides, GRIM lets the programmer to define custom policies for adapting the model to fit specific application requirements.

The next subsection takes a deeper look at the access methods provided by GRIM for agent-resource interaction. Then, subsections 4.2.2 and 4.2.3 discuss the policy support of GRIM at the middleware-level and application-level, respectively.

4.2.1 Accessing resources

Unlike its predecessor RMF [52], 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 [55]. Here are two 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 [6] from the remote host, instead of moving the agent to the host. This approach saves bandwidth at the cost of using more storage space on the local host.
- The requested resource is a remotely-callable resource, such as a Web Service, where a transfer is usually not feasible. In this case, the proper way to use it is by remotely invoking the operations defined by the service, thus transferring only (potentially small) input arguments and results rather than the (usually heavier) 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 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. However, from the mobility point of view, we claim this is a generalized form of mobility since remote invocation implies control flow migra-

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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

tion, and replication can be considered as a form of resource migration. The three forms of mobility considered by GRIM, as illustrated in figure 8, are:

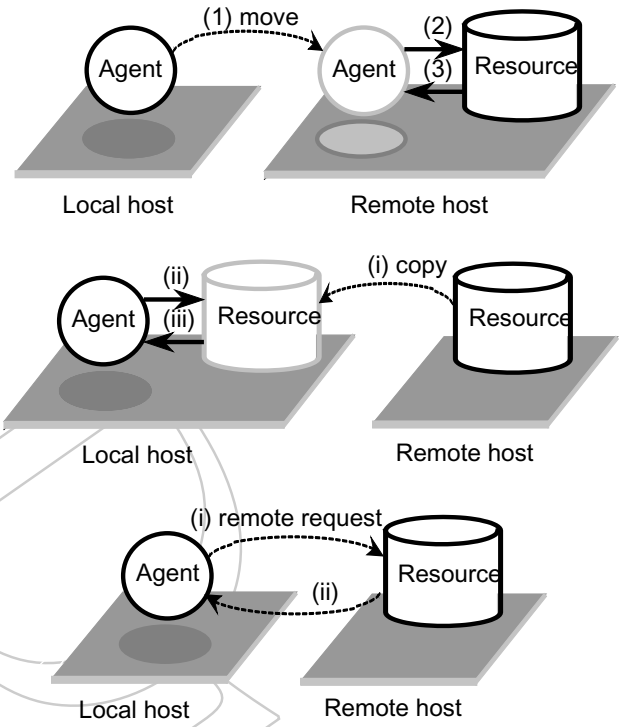


Fig. 8. Forms of mobility in GRIM: agent, resource and control flow migration

- *Agent migration (move)*: 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. At the middleware level, this could be supported by a *move* sentence implementing either a *strong* or *weak* agent migration mechanism [56]. *Strong* refers to the ability of a mobile agent run-time system to allow migration of both the code and the execution state of a mobile agent. In opposition, *weak* migration cannot transfer the execution state of a mobile agent. Current materializations of GRIM [52,53], including JGRIM, are based exclusively on strong migration.
- *Resource migration (fetch)*: Transfers the resource from the hosting location to the current agent's location, by copying it to a shared repository accessible to the agent. Resources such as data streams, structured files and code 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* operation, which implements the logic along with all technology-related issues for moving that resource between two hosts. At present, we have implemented this support in JGRIM 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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

execution flow. Technologies like 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 Web Services Invocation Framework (WSIF), a Java API that allows to stubless and dynamically call Web Services based upon the examination of WSDL descriptions at run-time.

WSIF enables developers to interact with abstract representations of Web services through their WSDL descriptions instead of working directly with the Simple Object Access Protocol (SOAP) APIs, which is the usual programming model. With WSIF, developers can work with the same programming model regardless of how the Web service is implemented and accessed.

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; fixed resources represent data and software components whose transfer is either not suitable (e.g. a very large database) or not allowed (e.g. a password-protected file). Figure 9 summarizes this taxonomy.

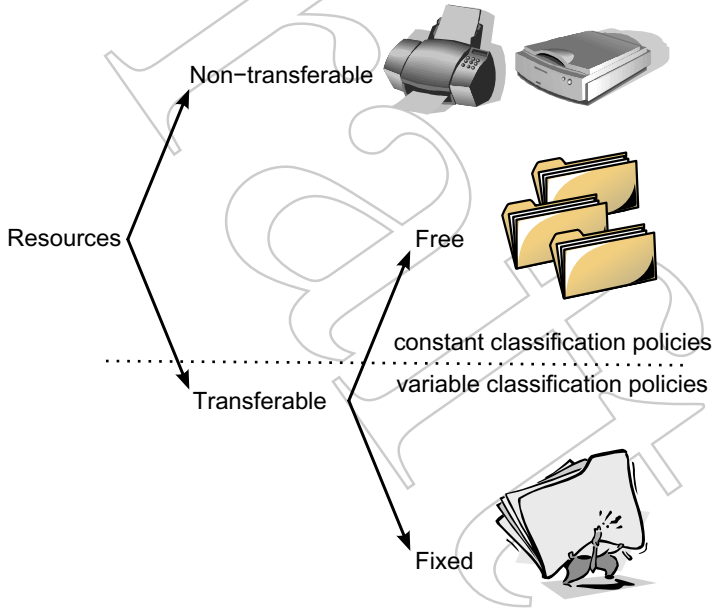


Fig. 9. GRIM resource taxonomy

Transferability of free and non-transferable resources is determined statically ("always" and "never", respectively). On the other hand, transferability of fixed resources is determined at run-time, even when *by nature* this kind of resources could be transferred. GRIM defines the concept of *classification policy*, which comprises

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

the logic in charge of dynamically associating a resource with a single leaf from the taxonomy tree. Resource publishers must provide a classification policy for every resource they add to a host. Classification policies can be either *constant*, where the resource type is configured statically and do not vary along time (non-transferable and free transferable), or *variable*, where the type is computed dynamically.

4.2.2 Middleware-level policies

Deciding what form of mobility to use when accessing a resource strongly depends on highly dynamic execution conditions such as agent and resource size, network metrics, available processing power and storage space, etc. In this sense, *middleware-level policies* (m-policies for short) allows hosts to 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 unstable link should trigger agent mobility". This policy will affect the behavior of 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 to contact some or all of them for interacting with their hosted resource. In addition, 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 when binding. In other words, m-policies provide criteria to select which resource to use from a set of candidates exposing the same protocol.

A major 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.

4.2.3 Agent-level policies

GRIM allows agents to delegate decisions about from where and how to access resources, but doing so implies that developers lose control of how these decisions are actually made. In some cases, this may be troublesome. 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 the agent has a significant size, and an m-policy for optimizing network traffic has been configured for M_1. As a result, every time the

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

agent accesses R, a trap will be triggered, and then a remote query to the database will be 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, bandwidth usage will be good or poor depending on the number of database queries (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 a 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 declared separately from the agent's implementation and 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 access method that will be used. A-policies have a greater priority than m-policies, in the sense that agent decisions overrides middleware ones for the same protocol.

To clarify these ideas, we will define a policy to overcome the situation described before in the context of 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 trA = pr.profile("latency", "localhost", hostA);
    double trB = pr.profile("latency", "localhost", hostB);
    return (trA < trB)? hostA : hostB;
  }
}
```

The `accessFrom` method defines the logic to select the desired resource instance from any pair of candidates, whereas `accessWith` contains the behavior for choosing an access strategy given any pair of valid access methods for the database. The former returns the *move* method whatever the candidates are, thus forcing the agent to migrate to M_2 upon the first query to the database D. Now let us suppose that D would be replicated 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 Network Weather Service [57], a distributed system that monitors and forecasts the performance of a network and its nodes.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

Policies can also redefine two more methods, not shown in the previous example, named `beforeCallPolicy` and `afterCallPolicy`. 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 could implement a stateful policy taking decisions based on statistical methods or learning algorithms.

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 in some cases, thus the mechanism must be "tamed". In fact, according to the Merriam-Webster Dictionary [7], "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.

5 Evaluation and discussion

In this section we report a comparison between JGRIM, Proactive and Ibis, two Java-based platforms for developing Grid applications. Basically, we separately used these three alternatives to gridify an existing implementation of the k-NN algorithm presented in past sections. Then, we took metrics on the resulting code to qualitatively analyze how hard is to port a Java application to the Grid with either of the three alternatives.

It is worth mentioning that the purpose of the experiments presented in this section is not to evaluate performance, but to measure the effort to gridify an existing application, and the impact of this process on the application code. In fact, in previous sections we have discussed the mechanisms our approach provides to address efficiency issues. However, evaluating how effective these mechanisms are for achieving good performance are out of the scope of this paper.

The original version of the k-NN algorithm was implemented as a single class accessing a file-based dataset through a `Dataset` class. This component includes a method for reading block of items, and methods for obtaining the size and the number of dimensions of the dataset. For the sake of experimenting on a simple Grid-like environment, the dataset was wrapped with a Web Service exposing analogous operations. 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

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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

algorithm itself, plus the code for interacting with the dataset, performing exception handling and parallelization.

- *PLOC (Platform-specific Lines of Code)*: This metric counts the number of source code lines which access the underlying platform API by using object types or calling their methods. The larger the value of PLOC, the more the level of tying between the application code and the platform API. Clearly, it is desirable to keep PLOC as low as possible, in order to avoid making applications dependent on the platform and hinder their portability to other Grid platforms.
- *NOC (Number Of Classes) and NOI (Number Of Interfaces)*: They represent the number of application classes and interfaces, respectively, without taking into account platform-specific or third-party ones, or compilation units being part of the dataset Web Service implementation.
- *NOT (Number Of Types)*: It simply counts the number of object types (classes and interfaces) which are referenced from within any of the compilation units of the application. It can be viewed as the sum of NOC and the number of classes/interfaces used after the Java reserved keywords *extends*, *implements* or *import*. 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.
- *LC/GC (Logic Code/Grid Code ratio)*: It represents the percentage of code lines that corresponds to pure application logic. Grid code takes into account the source lines either added or modified to the original implementation of the algorithm in order to call operations of the dataset Web Service, deal with errors, and adapt the application code to the underlying Grid platform. The LC/GC value for the original version of the application is 100%, that is, there is no code related to Grid behavior at all.
- *Bytecode information*, given by the number of generated .class files, and the total size (in Bytes) of these files.

Table 1 shows the resulting metrics for each one of the four variants of the application (Original, Ibis, Proactive and JGRIM). Figure 11 summarizes TLOC and LC/GC metrics, and Figure 12 shows the overhead incurred in gridifying the application in terms of source code lines. In order to perform a fair comparison, the Java code was pre-formatted and the imports were optimized by using the source code tools provided by the Eclipse IDE. Besides, all versions, including the original, were implemented by the same person.

Figure 10 shows simplified class diagrams for the Ibis, Proactive and JGRIM versions of the gridified k-NN algorithm. In the latter case, the tasks of extending the `JGRIMAgent` class, adding setters/getters, and realizing `DatasetInterface` and `SpawnInterface` are performed automatically based on the dependencies declared by the developer.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

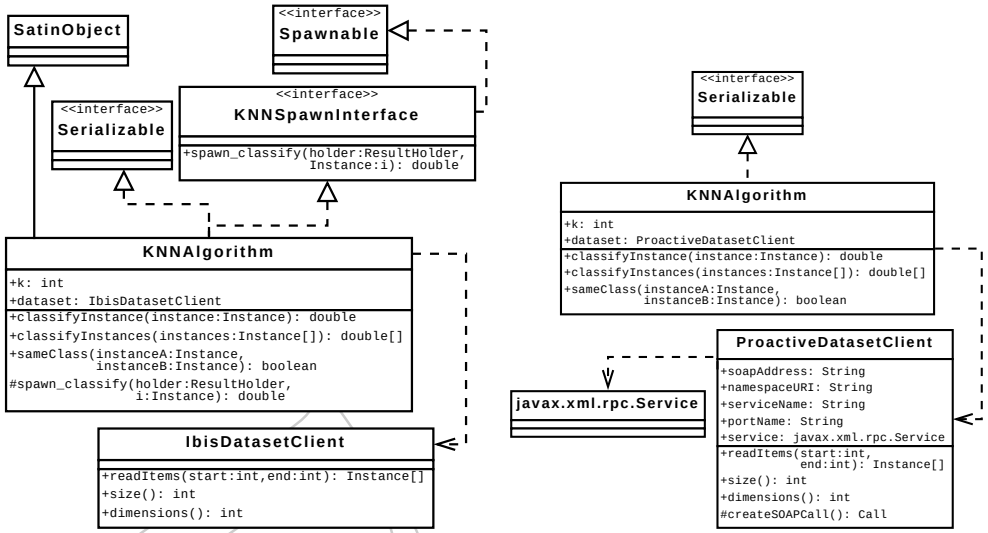


Fig. 10. Gridification of the KNN algorithm: simplified class diagrams

Table 1
Gridification of the k-NN algorithm: code metrics

k-NN Version	TLOC	PLOC	NOC	NOI	NOT	LC/GC	# .class	Size
Original	172	–	4	0	11	100% (172/172)	4	7257
Ibis	1263	5	25	2	77	8.87% (112/1263)	35	83131
Proactive	259	10	4	0	26	47.9% (124/259)	5	13270
JGRIM	147	0	3	2	9	85.0% (125/147)	5	13952

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

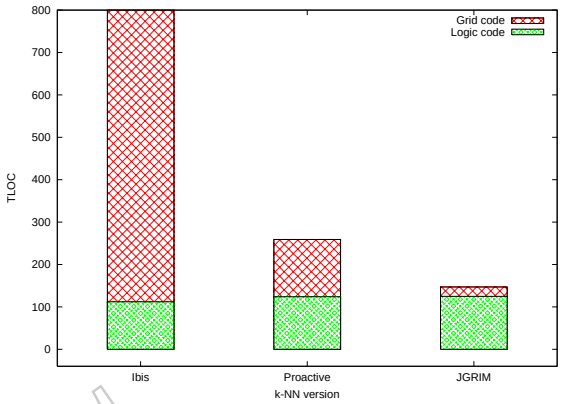


Fig. 11. TLOC and LC/GC after gridification

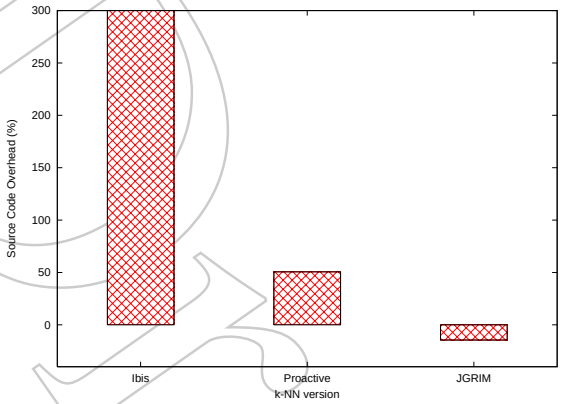


Fig. 12. Source code overhead introduced by the gridification process

The size of the application for the case of Ibis was 1263 lines (seven times bigger than the original implementation). Only a small percentage of the code (less than 9%) resulted in pure application logic, since we had to provide a lot of code mostly to generate and use a client-side proxy [8] to the dataset Web Service. As a consequence, NOC and NOT also suffered (six and seven times bigger, respectively), since more application classes/interfaces were created, and also extra APIs for low-level interaction with services were imported. The code for the proxy was tied to our dataset service WSDL description, therefore lacking reusability as the information for invoking services on the Grid (e.g. the WSDL location) usually changes over time. Surprisingly, despite being a platform proposed for Grid development, Ibis does not offer any facilities for using Web Services.

[8] The proxy was built using Sun's Java Web Services Development Pack 2.0

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

The Proactive version of the application showed an LC/GC value near to 48%. Little changes were made to the original code. The PLOC was slightly greater than for the Ibis case (10 lines, less than 4%). However, we faced two main problems when gridifying with Proactive. First, Proactive also lacks full support for using Web Services, as it only offers a set of classes which allow applications to interact with either SOAP-enabled services or 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 our dataset was initially wrapped with a non-SOAP Web Service, we needed to implement an active object for interfacing the data, expose it as a SOAP service, and finally write a client to use it. Second, we had to change the return type of the `classifyInstance` method, wrapping its result with a Proactive core API class in order to support parallelization. As a consequence, the interface of the gridified application differs from the original implementation, containing non-standard datatypes and interaction conventions. This, in turn, makes application discovery within a service-oriented Grid more difficult, as details on required service interfaces must usually be provided by clients for the discovery process to be effective.

The gridification of the k-NN algorithm with JGRIM resulted in 147 lines of code (plus 31 lines of XML configuration automatically generated by our gridifying tool). The code was even smaller than the non-gridified version, since service interaction and exception handling is now managed at the middleware level. Besides, the application is not tied to the platform at all, since none of the JGRIM API object types were used. However, the most interesting aspect of the implementation is that only four lines of code were manually modified, in order to interact with the dataset by means of a service dependency and parallelize the `classifyInstances` method. Only 10 lines were added since we created two Java interfaces for representing dependencies (i.e. dataset and method spawning). The resulting code was very clean and easy to understand. In addition, making and testing further improvements over the algorithm outside our Grid setting is straightforward, since we can easily “inject” another source for the dataset to the application without modifying its code. On the Ibis and Proactive versions, however, this is not the case, since a significant part of the application must be rewritten/discarded if another technology for interfacing the dataset is employed.

Another interesting result of our evaluation is concerned with the size of the Java bytecode. After source code gridification and compilation, the binary size of the JGRIM and Proactive implementations are about 13 KB, versus 81 KB for the Ibis version. In this latter case, though some functionality for interacting with the dataset was added to the application (resulting in 55 KB of bytecode), the final deployment generated a lot of .class files for managing both parallelization and 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 roughly 500% more bandwidth than either the

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

JGRIM or Proactive implementation. Note that, for more complex applications, this difference could be even bigger.

As shown in Table 1, the bytecode size of the Proactive solution was slightly smaller than the one 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 run-time, thus this overhead is not present in the above results because it is very difficult to measure. On the other hand, we found that a big percentage of bytecode for the JGRIM implementation (7 KB out of 13 KB) were instructions specifically targeted for dealing with thread-level serialization and migration. In fact, the binary size without this support was even smaller than the compiled version of the original k-NN implementation (6275 Bytes vs. 7257 Bytes). In this sense, in order to make binary code lighter and more compact, at least in appearance, a technique for instrumenting bytecode similar to the one used by Proactive is being implemented. The idea is to develop a special class loader which instruments applications at run-time, thus dynamically enabling them for being mobile.

Finally, a limitation of the JGRIM solution was concerned with parallelization. As explained at the beginning of this paper, our method spawning technique works by blocking the MGS the first time it requests the result of an unfinished spawned computation. As a consequence, the technique is by far less effective when the same method is called inside a loop control structure which accesses the result of a call before another call takes place, such as the case of the `classifyInstances` method. Conversely, on the Ibis and Proactive versions of the application we were able to fully take advantage of parallelization for `classifyInstances` by explicitly spawning several calls to `classifyInstance`. In this way, significant performance benefits might be obtained when classifying different instances concurrently at several hosts, at the cost of having an extra implementation effort since more changes to the original application are needed.

The situation described before is a clear example of a common tradeoff likely to be found when parallelizing an application: ease of programming versus the flexibility to control the application execution [58]. 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 run-time system. On the other hand, languages based on explicit parallelism aim at supplying constructs or APIs for describing the way in which 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.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

Middlewares like Proactive and Ibis, which are inherently performance-oriented, are designed to provide explicit parallelization. The programmer has a finer control of parallelism, which in turn makes gridification harder. Conversely, JGRIM is based on implicit parallelism, since it provides parallelization facilities in those places of an application where it can be transparently used (i.e. without coding effort). Unlike Proactive or Ibis, our approach is more suited for users wanting to effortlessly get their applications running on the Grid.

6 Conclusions

Grid Computing is a new paradigm for distributed computing whose purpose is to transparently offer applications a single "supercomputer" by virtualizing all computational resources within a computer network. Unfortunately, Grid Computing has not yet delivered its full potential due to the lack of platforms and tools for easy development, deployment and tuning of applications in this new heterogeneous and complex environment.

Mobile agents is a popular technology for materializing distributed applications. Particularly, the Grid can largely benefit from mobile agents both for application development and service provisioning. We propose JGRIM, a Java platform for mobile agent-based programming on the Grid. The main goal of JGRIM is to simplify the gridification of applications by hiding the inherently complex nature of the Grid as much as possible. The JGRIM programming model is very easy to use and tune, and it is particularly well suited for building service-oriented applications.

We have shown the practical advantages of JGRIM through code metrics. As reported, JGRIM allows for a better separation of application logic and code for common Grid functionality such as service discovery and invocation, thus making the task of writing and porting applications to the Grid more easy. Besides, custom decisions for tuning Grid applications can be specified separately from the logic, therefore letting developers to seamlessly adapt and optimize the same application to various Grid environments and settings. Despite those encouraging results, we also expect to conduct experiments in order to test the various performance aspects of JGRIM applications with respect to related approaches. We are currently developing a number of prototype applications and Grid services to test our ideas in a realistic Grid setting. Preliminary tests conducted on a LAN showed that JGRIM applications incur in a performance overhead of 5-10% compared to their non--gridified counterparts. Further improvements on our parallelization and tuning support should dramatically reduce this overhead.

A major drawback of our approach is the lack of security provisioning. Since JGRIM applications can travel across boundaries of different administrative domains looking for Grid services, security is crucial to guarantee the integrity of

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

both applications and resource providers. The main factor that has hindered the widespread adoption of mobile agents is precisely their security problems [59]. Nevertheless, the problem of supporting security in mobile agent systems has been widely acknowledged and extensively investigated [60–62]. In this sense, a line of future research is to incorporate proper security mechanisms into our agent execution model, and also to study how these mechanisms can be integrated with existing security services for the Grid such as GSI [9] .

Another limitation of JGRIM arises as a consequence of the assumptions made when gridifying applications. Specifically, JGRIM assumes those applications as being constructed under a component-based paradigm (e.g. JavaBeans), comprising a number of components exposing and requesting services according to well-defined interfaces. However, this assumption does not likely hold for all kind of applications, specially for legacy Java code. Fortunately, the problem of componentizing legacy object-oriented applications has been addressed already in the literature [63,64]. We are planning to apply a similar approach to our gridification process.

We are also enhancing the parallelization support of JGRIM. First, our method spawning scheme is being extended to transparently parallelize the invocation of any Grid service within an application besides internal operations. Second, we are adding facilities to our policy support for customizing parallelization, such as controlling at run-time the way services are called (synchronously vs. asynchronously), or under what conditions the execution of a certain operation should be spawned. An interesting recent work in this line is POP-C++ [65], a parallel programming language based on C++ that provides inter-object and intra-object parallelism according to various method invocation semantics. Third, we are exploring the viability of integrating JGRIM with existing job submission and execution services such as those provided by Ibis or Globus's GRAM. The idea is to transparently spawn and delegate the execution of a cycle-consuming method to middlewares suited for running CPU-intensive applications, therefore also leveraging useful features like load balancing, execution monitoring and fault tolerance.

Finally, we are working on programming tools to make JGRIM easy to adopt and use. Specifically, we are developing a plug-in for the Eclipse SDK, which will help programmers to gridify a Java application by graphically defining dependencies, and creating/associating custom policies to them.

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

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper. We also thank Marco Crasso and Pablo Gotthelf for their valuable comments.

References

[1] I. Foster, C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, San Francisco, CA, USA, 2003.

[2] I. Foster, The Grid: Computing without bounds, Scientific American 288 (4) (2003) 78–85.

[3] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organization, The International Journal of High Performance Computing Applications 15 (3) (2001) 200–222.

[4] L. F. G. Sarmenta, S. Hirano, Bayanihan: Building and studying volunteer computing systems using java, Future Generation Computer Systems, Special Issue on Metacomputing 15 (5-6) (1999) 675–686.

[5] A. Natrajan, M. A. Humphrey, A. S. Grimshaw, The Legion support for advanced parameter-space studies on a grid, Future Generation Computer Systems 18 (8) (2002) 1033–1052.

[6] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne, A worldwide flock of condors: Load sharing among workstation clusters, Future Generation Computer Systems 12 (1) (1996) 53–65.
URL http://www.sciencedirect.com/science/article/B6V06-3VTJ%HKC-6/2/31dc5a07415344096b61120f4d0109b3

[7] D. Thain, T. Tannenbaum, M. Livny, Condor and the grid, in: F. Berman, G. Fox, A. Hey (Eds.), Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc., New York, NY, USA, 2003, pp. 299–335.

[8] I. Foster, Globus toolkit version 4: Software for Service-Oriented systems, in: IFIP International Conference on Network and Parallel Computing, Vol. 3779, Springer-Verlag GmbH, 2005, pp. 2–13.

[9] H. Stockinger, Defining the grid: A snapshot on the current view, Journal of SupercomputingTo appear.

[10] W3C Consortium, Soap version 1.2 part 0: Primer, W3C Recommendation, http://www.w3.org/TR/soap12-part0/ (Jun. 2003).

[11] W3C Consortium, Web services description language (wsdl) version 2.0 part 1: Core language, W3C Candidate Recommendation, http://www.w3.org/TR/wsdl20/ (Mar. 2006).

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

[12] OASIS Consortium, Uddi version 3.0.2, UDDI Spec Technical Committee Draft, `http://uddi.org/pubs/uddi_v3.htm` (Oct. 2004).

[13] OGSA-WG, Defining the grid: A roadmap for OGSA standards, `http://www.gridforum.org/documents/GFD.53.pdf` (Sep. 2005).

[14] 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` (May 2006).

[15] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, J. Webber, Web service grids: an evolutionary approach: Research articles, Concurrency and Computation: Practice and Experience 17 (2-4) (2005) 377–389.

[16] C. Chapman, P. Wilson, T. Tannenbaum, M. Farrellee, M. Livny, J. Brodholt, W. Emmerich, Condor services for the global grid: interoperability between Condor and OGSA, in: Proceedings of 2004 UK e-Science All Hands Meeting, Nottingham, UK, 2004, pp. 870–877.

[17] C. Chapman, C. Goonatilake, W. Emmerich, M. Farrellee, T. Tannenbaum, M. Livny, M. Calleja, M. Dove, Condor BirdBath: Web service interfaces to Condor, in: Proceedings of 2005 UK e-Science All Hands Meeting, Nottingham, UK, 2005, pp. 737–744.

[18] M. Chetty, R. Buyya, Weaving computational grids: How analogous are they with electrical grids?, Computing in Science and Engineering 4 (4) (2002) 61–71.

[19] I. J. Taylor, From P2P to Web Services and Grids: Peers in a Client/Server World, Computer Communications and Networks, Springer, 2005.

[20] J. Kommineni, 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, M. Bubak (Eds.), Advances in Grid Computing - EGC 2005, Vol. 3470 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 961–971.

[21] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, P. Kacsuk, GEMLCA: Running legacy code applications as grid services, Journal of Grid Computing 3 (1-2) (2005) 75–90.

[22] Q. Ho, T. Hung, W. Jie, H. Chan, E. Sindhu, S. Ganesan, T. Zang, X. Li, GRASG - A Framework for 'gridifying' and running applications on service-oriented Grids, in: Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), IEEE Computer Society, 2006, pp. 305–312.

[23] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Grid Computing: Software Environments and Tools, Springer-Verlag, 2006, Ch. Programming, Deploying, Composing, for the Grid, pp. 205–229.

[24] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H. E. Bal, Ibis: a flexible and efficient Java based grid programming environment, Concurrency and Computation: Practice and Experience 17 (7-8) (2005) 1079–1107.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

[25] A. Jugravu, T. Fahringer, JavaSymphony, a programming model for the grid, Future Generation Computer Systems 21 (1) (2005) 239–247.

[26] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, A. Slominski, On building parallel and grid applications: Component technology and distributed services, Cluster Computing 8 (4) (2005) 271–277.

[27] S. Venugopal, R. Buyya, L. Winton, A grid service broker for scheduling e-science applications on global data grids, Concurrency and Computation: Practice and Experience 18 (6) (2006) 685–699.

[28] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, K. Verstoep, Programming environments for high-performance grid computing: the albatross project, Future Generation Computer Systems 18 (8) (2002) 1113–1125.
URL http://www.sciencedirect.com/science/article/B6V06-46H4%DN9-1/2/e30de0e157088a443b228382a9a86f47

[29] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, H. E. Bal, Satin: Simple and efficient Java-based grid programming, Scalable Computing: Practice and Experience 6 (3) (2005) 19–32.

[30] D. Abramson, J. Giddy, L. Kotler, High performance parametric modeling with nimrod/g: Killer application for the global grid?, in: 14th International Symposium on Parallel and Distributed Processing (IPDPS '00), IEEE Computer Society, Washington, DC, USA, 2000, p. 520.

[31] M. O. Neary, B. O. Christiansen, P. Cappello, K. E. Schauser, Javelin: Parallel computing on the internet, Future Generation Computer Systems 15 (5–6) (1999) 659–674.
URL http://www.sciencedirect.com/science/article/B6V06-405T%F92-D/2/87f701cab290749aa3b04ecf06c2470d

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

[33] Globus Alliance, The java cog kit, http://wiki.cogkit.org/index.php/Java_CoG_Kit (last accessed January 2007).

[34] E. Seidel, G. Allen, A. Merzky, J. Nabrzyski, Gridlab–a grid application toolkit and testbed, Future Generation Computer Systems 18 (8) (2002) 1143–1153.
URL http://www.sciencedirect.com/science/article/B6V06-46V4%5YH-4/2/0baa0d451554a5afda4812b708306a71

[35] IBM alphaWorks, Grid application framework for java, http://alphaworks.ibm.com/tech/GAF4J (Oct. 2004).

[36] A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora, M. Pathak, Design of the ajanta system for mobile agent programming., Journal of Systems and Software 62 (2) (2002) 123–140.

[37] D. B. Lange, M. Oshima, Seven good reasons for mobile agents, Communications of the ACM 42 (3) (1999) 88–89.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

[38] B. Di Martino, O. F. Rana, Grid performance and resource management using mobile agents, in: V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller (Eds.), Performance analysis and grid computing, Kluwer Academic Publishers, Norwell, MA, USA, 2004, pp. 251–263.

[39] F. Ishikawa, N. Yoshioka, Y. Tahara, S. Honiden, Towards synthesis of web services and mobile agents, in: Z. Maamar, C. Lawrence, D. Martin, B. Benatallah, K. Sycara, T. Finin (Eds.), Workshop on Web Services and Agent-Based Engineering (WSABE) (AAMAS'2004), New York, NY, USA, 2004.

[40] R. M. Barbosa, A. Goldman, MobiGrid: Framework for mobile agents on computer grid environments, in: Mobility Aware Technologies and Applications, Vol. 3284 of Lecture Notes in Computer Science, Springer Berlin-Heidelberg, 2004, pp. 147–157.

[41] M. Fukuda, K. Kashiwagi, S. Kobayashi, Agentteamwork: Coordinating grid-computing jobs with mobile agents, International Journal of Applied Intelligence, Special Issue on Agent-Based Grid Computing 25 (2) (2006) 181–198.

[42] A. Suna, G. Klein, A. El Fallah-Seghrouchni, Using mobile agents for resource sharing, in: IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04), IEEE Computer Society, Los Alamitos, CA, USA, 2004, pp. 389–392.

[43] L. Chunlin, L. Layuan, Apply agent to build grid service management, Journal of Networks and Computer Applications 26 (4) (2003) 323–340.

[44] R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque, A resource discovery service for a mobile agents based grid infrastructure, in: High performance scientific and engineering computing: hardware/software support, Kluwer Academic Publishers, Norwell, MA, USA, 2004, pp. 189–200.

[45] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, J. Shalf, The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment, The International Journal of High Performance Computing Applications 15 (4) (2001) 345–358.
URL `citeseer.ist.psu.edu/article/allen01cactus.html`

[46] S. Vadhiyar, J. Dongarra, Self adaptability in grid computing, Concurrency and Computation: Practice and Experience, Special Issue on Grid Performance 17 (2–4) (2005) 235–257.

[47] J. Hunter, W. Crawford, Java Servlet Programming, O'Reilly & Associates, Inc., 1998.

[48] Apache Software Foundation, Jakarta Commons Javaflow, `http://jakarta.apache.org/commons/sandbox/javaflow` (Jun. 2006).

[49] R. Johnson, J2EE development frameworks, Computer 38 (1) (2005) 107–110.

[50] C. Walls, R. Breidenbach, Spring in action, Manning, 2005.

[51] B. V. Dasarathy, Nearest neighbor (NN) norms: Nn pattern classification techniques, IEEE Computer Society Press Tutorial, 1991.

This is a preprint of the article: "JGRIM: An approach for easy gridification of applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 24, Num. 2, pp. 99-118. 2008. doi:10.1016/j.future.2007.04.011"

The final publication is available at http://dx.doi.org/10.1016/j.future.2007.04.011

[52] A. Zunino, C. Mateos, M. Campo, Reactive mobility by failure: When fail means move, Information Systems Frontiers. Special Issue on Mobile Computing and Communications 7 (2) (2005) 141–154, ISSN 1387-3326.

[53] C. Mateos, A. Zunino, M. Campo, Extending movilog for supporting web services, Computer Languages, Systems & Structures 31 (1) (2007) 11–31.

[54] P. Gotthelf, M. Mendoza, A. Zunino, C. Mateos, Gmac: An overlay multicast network for mobile agents, in: Proceedings of the 6^{th} Argentinian Symposium on Technology (ASAI 2005 - 34^{th} JAIIO), SADIO, Rosario, Santa Fé Argentina, 2005.

[55] A. Zunino, C. Mateos, M. Campo, Enhancing agent mobility through resource access policies and mobility policies, in: V ENIA, XXV Congresso da Sociedade Brasileira de Computacão (SBC), San Leopoldo, RS, Brasil, 2005, pp. 851–860.

[56] A. Fuggetta, G. P. Picco, G. Vigna, Understanding code mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 342–361.

[57] R. Wolski, N. Spring, J. Hayes, The network weather service: A distributed resource performance forecasting service for metacomputing, Future Generation Computer Systems 15 (5-6) (1999) 757–768.

[58] V. W. Freeh, A comparison of implicit and explicit parallel programming, Parallel and Distributed Computing 34 (1) (1996) 50–65.

[59] D. Kotz, R. S. Gray, Mobile agents and the future of the Internet, ACM Operating Systems Review 33 (3) (1999) 7–13.

[60] W. A. Jansen, Countermeasures for mobile agent security, Computer Communications 23 (17) (2000) 1667–1676.

[61] J. Claessens, B. Preneel, 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) (2003) 28–48.

[62] P. Bellavista, A. Corradi, C. Federici, R. Montanari, D. Tibaldi, Security for mobile agents: Issues and challenges, in: I. Mahgoub, M. Ilyas (Eds.), Handbook of Mobile Computing, CRC Press, 2004, Ch. 39, pp. 941–958.

[63] E. Lee, B. Lee, W. Shin, C. Wu, A reengineering process for migrating from an object-oriented legacy system to a component-based system, in: 27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, IEEE Computer Society, 2003, pp. 336–341.

[64] S. D. Kim, S. H. Chang, A systematic method to identify software components, in: 11th Asia-Pacific Software Engineering Conference (APSEC 2004), IEEE Computer Society, 2004, pp. 538–545.

[65] T. Nguyena, P. Kuonen, Programming the grid with pop-c++, Future Generation Computer Systems 23 (1) (2007) 23–30.