Software: Practice and Experience

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

m-JGRIM: A Novel Middleware for Gridifying Java Applications

Journal:	*Software: Practice and Experience*
Manuscript ID:	draft
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	
Complete List of Authors:	Mateos Diaz, Cristian; ISISTAN - UNCPBA, Computación y Sistemas Zunino, Alejandro; ISISTAN - UNCPBA, Computación y Sistemas Campo, Marcelo; ISISTAN - UNCPBA, Computación y Sistemas
Keywords:	Grid Computing, Gridification, Grid middlewares, JGRIM, Dependency Injection, Java

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **00**:1–40 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

m-JGRIM: A Novel Middleware for Gridifying Java Applications

Cristian Mateos[1,2,*], Alejandro Zunino[1,2] and Marcelo Campo[1,2]

[1] *ISISTAN Research Institute - UNICEN*
[2] *Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)*

SUMMARY

The benefits of Grids for building massively distributed applications have been broadly acknowledged. However, the high complexity of developing Grid applications compromises the widespread adoption of the paradigm. In a previous article [1], we described JGRIM, a method for easily "gridifying" component-oriented Java applications, which is based on non-invasively injecting Grid functionality into ordinary code through Dependency Injection. In this paper, we briefly revisit JGRIM and present m-JGRIM, a novel Java middleware that materializes JGRIM concepts. We also provide an evaluation of the performance of m-JGRIM. Grid practitioners should find this paper useful in having an assessment of the practical benefits and costs of gridifying applications with the middleware, and a down-to-earth description of JGRIM, whose advantages for Grid-enabling applications from a software engineering perspective have been already evaluated.

KEY WORDS: Grid Computing, Gridification, Grid middlewares, JGRIM, Dependency Injection, Java

1. INTRODUCTION

Grid Computing is a paradigm for distributed computing based on virtualizing resources in a network to execute resource-intensive applications. Such an arrangement is called a *Grid* [2]. Typically, Grid applications are intended to solve scientific or engineering problems that require a large number of computational resources such as CPU cycles, memory, network bandwidth and data. Examples of such applications include protein folding, financial modeling and climate simulation.

The first attempts to establish Grids focused on supporting CPU-intensive, large-scale applications by linking supercomputers [2]. By then, Grids were mostly local networks that linked together powerful and dedicated computers. With the inception of Internet standards, Internet-wide Grids and then applications such as SETI@home [3] and Evolution@home [4] came into existence. At this stage, there was not a solid idea of Grid resource *virtualization* yet and implementing Grid applications required

*Correspondence to: Cristian Mateos (cmateos2006@gmail.com), ISISTAN Research Institute, UNICEN. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 43-9682. Fax.: +54 (2293) 43-9681.

Received
Revised

to programmatically access the underlying infrastructure. Few years later, the first Grid middlewares appeared (e.g. Globus [5], Condor [6] and Legion [7]). The goal of these technologies is twofold: to virtualize Grid resources by means of *services* (e.g. job scheduling/balancing, resource brokering, data movement, security, etc.) and to supply developers with rich APIs for using these services.

Recently, besides delivering more Grid services to applications, research in Grid middlewares has emphasized on both *interoperability* and *consumability* of the services. Efforts in the former category have yielded as a result Grid standards such as OGSA and WSRF [8]. These standards strongly rely on Web Service technologies, which provide an adequate solution to the problem of heterogeneous systems integration across administrative domains [9, 10]. This has motivated the evolution of Grid middlewares to new versions based on Web Services. Complementary, researchers have been looking for better development tools to simplify the consumption of Grid services from within user applications. These tools seek to provide facilities to let developers to benefit from middleware services with little (ideally zero) code provisioning. These efforts are grouped into programming toolkits and gridification methods [11]. The former provide high-level APIs that abstract away the details to use middleware services (e.g. [12, 13, 14]). By using these tools, Grid programming is done at a higher level of abstraction, thus less code and effort compared to directly using middleware APIs is required. However, as they are programming facilities, these approaches usually assume that developers are proficient on the toolkit being used. Alternatively, gridification methods (e.g. [15, 16, 17, 18]) allow developers to easily incorporate Grid services into existing codes. Thus, these methods are intended to support users having little and ideally no background on Grid programming.

In [1], we described JGRIM, a gridification method that targets component-based Java applications. Conceptually, JGRIM works by non-invasively injecting Grid services into ordinary codes. With JGRIM, developers can focus on the development and testing of application logic without worrying about common Grid concerns such as resource discovery and execution management. At the middleware level, these concerns are materialized by *metaservices*, which represent existing Grid services. Metaservices are then "injected" by a JGRIM-compliant middleware that provides a runtime environment for gridified applications. This paper describes a proof-of-concept materialization of JGRIM named m-JGRIM, focusing on explaining its metaservice injection capabilities. We evaluated m-JGRIM at the macro level by comparing it with Ibis [15] and ProActive [16], two well-established Java-based Grid libraries. The experiments were carried out by measuring code quality, execution time and network usage of two Grid applications on a WAN. We also evaluated the cost of performing injection of various m-JGRIM metaservices to get an insight on the penalty of gridifying applications with m-JGRIM at the micro level. For injecting metaservices, m-JGRIM extensively modifies the anatomy of ordinary applications by intercepting interactions between their components and altering their bytecode, which may impact on the performance of transformed applications. This assessment quantifies this impact and provides guidelines for developing Grid applications with m-JGRIM.

The motivation for writing this paper stems from the fact that, in spite of the engineering advantages of JGRIM as a gridification method [1], materializing its concepts raises a number of difficult issues from a technological standpoint. Two of the most challenging aspects in this regard are how to support metaservice injection without code modification, which may involve the combined use of complex techniques such as bytecode instrumentation and aspect-oriented programming, and how to exploit existing Grid services at the middleware level and still achieve good performance. Therefore, besides offering a by-example explanation of the features of m-JGRIM for gridifying applications, this paper details how the middleware materializes such aspects.

The rest of the paper is organized as follows. The next section discusses related works and explains how m-JGRIM improves over them. Section 3 overviews JGRIM. Section 4 describes m-JGRIM. Section 5 reports the evaluation of m-JGRIM. Section 6 concludes the paper.

2. RELATED WORK

Research in Grid middlewares has experienced a substantial growth in the last years. A noticeable fact is the way Java has influenced this growth due to its "write once, run anywhere" philosophy, which promotes platform independence. Besides offering APIs for programming Grid applications, many of these middlewares actually materialize methods for gridification of Java software.

ProActive [16] is a Grid middleware that provides *technical services*, which allows users to address non-functional concerns (mostly load balancing and fault tolerance) by plugging some configuration to applications at deployment time. Applications are built by composing mobile entities whose creation, migration and lookup must be performed programmatically. Likewise, JCGrid [19] supports distributed job scheduling for CPU-intensive tasks. In both cases, after gridifying an application, the application logic results mixed up with Grid-related code, rendering gridification and software maintenance thereafter difficult. Ibis [15] offers a Grid messaging library on top of which a variety of popular programming models are implemented (e.g. Satin [20] for parallelizing divide and conquer applications on WANs). Like ProActive and JCGrid, Ibis offers limited support for interoperable Grid protocols such as WSDL and UDDI. Similarly, GridGain [21] uses annotations to easily exploit distributed CPUs, but it does not target interoperability or pluggability of existing Grid services.

GMarte [22] is a high-level, object-oriented API on top of Globus services. With GMarte, users can compose and coordinate the execution of existing binary codes by means of a (usually small) new Java application. GMarte also features metascheduling and fault-tolerance via application-dependent checkpointing. However, as GMarte treats these codes as black boxes, their structure cannot be altered to better exploit Grids, this is, parallelize or distribute some portions of gridified codes. Similarly, XCAT-Java [23] supports execution of component-based applications as OGSA services on top of existing Grid middlewares (preferably Globus). Application components can also represent legacy binary programs. XCAT-Java provides an API for building complex applications by assembling service and legacy components. Though this task requires little coding effort, developers still have to programmatically manage component creation and linking. Besides, similar to GMarte, XCAT does not provide support for fine tuning components at the application level.

JavaSymphony [24] provides an execution model that semi-automatically deals with migration, parallelism and load balancing. These features can be also explicitly controlled through API primitives in the application code. Similarly, VCluster [25] supports efficient execution of parallel applications on SMP clusters, while Babylon [26] offers weak mobility, messaging and parallelism in an uniform API. Since these three middlewares are API-oriented Grid development tools, they require users to learn their API and perform extensive modifications on their non-gridified codes to use these APIs.

Our research started by conducting an exhaustive survey on approaches to easily "plug" applications to Grids [11]. One important finding from this recent study is that existing methods for gridifying software fall into two major categories: those that aim at separating application logic from Grid code (two-step gridifiers), and those that do not (one-step gridifiers). Particularly, two-step gridifiers are some way off from being effective tools for gridifying applications. Tools relying on an API-inspired

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

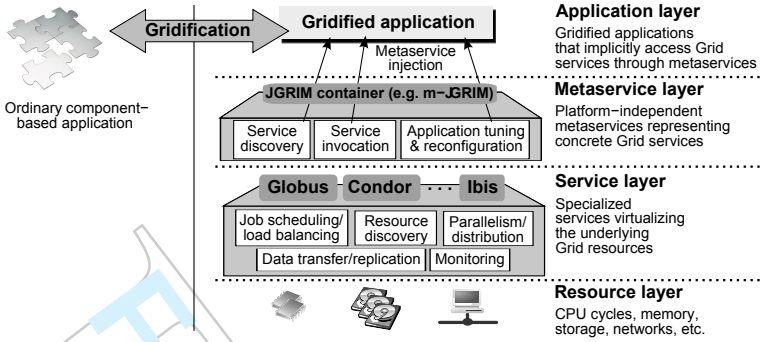


Figure 1: An overview of JGRIM

approach to gridification unavoidably require modifications to the code of the original applications, which in turn requires developers to learn Grid APIs and to put more effort into maintaining the gridified applications. Nevertheless, developers have more control of the internals of their applications. Moreover, tools based on gridifying by wrapping or composing existing applications (e.g. GMarte, XCAT-Java) simplify gridification, but prevent the usage of tuning mechanisms such as parallelism, mobility and distribution of individual application components. This represents a tradeoff between ease of gridification versus true flexibility to configure the runtime aspects of gridified applications [11].

m-JGRIM targets this tradeoff by avoiding excessive source modification when gridifying applications, yet offering means to effectively tune Grid applications. m-JGRIM preserves the integrity of the application logic by letting developers to concentrate on coding the functionality of applications, and then seamlessly adding Grid concerns to them. Unlike many of the above tools, its core API only have to be explicitly used when performing application tuning and, in such a case, the application logic is not affected. Because of the component based roots of its underlying gridification method (i.e. JGRIM), using m-JGRIM is similar to developing with popular Java models such as JavaBeans or EJB.

3. JGRIM

JGRIM is a method for easily porting applications to a Grid. JGRIM simplifies the development of Grid applications by separating the functional code from the code for accessing Grid services, which is non-intrusively and implicitly injected instead. Central to JGRIM is its semi-automatic *gridification process* that developers have to follow to gridify their applications. JGRIM accepts as input ordinary component-based applications, and transforms them to applications furnished with *metaservices*, which allow developers to access Grid services while minimizing the code modifications to interact with them. We refer to such transformed applications as *gridified* or *Grid-enabled* applications.

JGRIM (see Figure 1) adds a Metaservice layer that enables gridified applications to seamlessly use existing Grid middleware services, thus their execution is subject to a stack comprising four layers:

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

- *Resource*, which represents the physical infrastructure of a Grid.
- *Service*, which provides, by means of existing Grid middlewares, a catalog of services including resource brokering, job scheduling, parallelism, data management, etc. These services offer sophisticated Grid functionalities accessible to applications through specific APIs.
- *Metaservice*, which comprises metaservices that glue gridified applications and Grid services, isolating Grid-enabled applications from technology-related details for accessing the lower layer. A metaservice represents a set of Grid services offering similar functionality, namely:
  - *Service discovery*: The Service Discovery metaservice may talk, for example, to a UDDI registry [27] or the MDS-2 [28] to find a list of required Grid services, and to present the results to the upper layer in a uniform format. This metaservice hides all concrete Grid lookup services behind a technology-neutral lookup(serviceInterface) primitive.
  - *Service invocation*: Once an instance of a required Grid service is discovered, interaction with it comes next. This may involve to employ specific binding protocols and data type formats. Typically, these elements are specified in a WSDL descriptor [27], which represents the contact information of an individual Grid service. The goal of this metaservice is to provide a generic call(serviceDescriptor) primitive.
  - *Application tuning*: This metaservice is associated to certain application components to gain efficiency and robustness by leveraging existing Grid services for parallelism, load balancing and distribution. For example, all invocations on an embarrassingly parallel operation may be executed concurrently to improve performance and scalability. Similarly, a mission critical computation may be submitted to Globus, thus increasing fault tolerance. The metaservice also materializes *policies* [1, 29], this is, non-intrusive mechanisms by which developers can customize the way an application behaves within a Grid.
- *Application*, which contains Grid-enabled applications. During gridification, JGRIM enhances some of the original application components and their interactions via metaservices, so that at runtime some internal operation requests originated by components at this layer are handled by metaservices. Furthermore, m-JGRIM is the software artifact that provides an implementation for these metaservices.

JGRIM assumes that input applications are properly componentized, which is a common practice in Java development as evidenced by the popularity of component models such as JavaBeans or EJB [1]. This allows JGRIM to treat every application as a collection of interacting components, and enhancing some of these interactions with metaservices by using Dependency Injection (DI) [30].

3.1. Injecting metaservices into ordinary applications

DI achieves higher decoupling in component-based applications by enforcing components to be described through public interfaces, and reducing couplings by delegating the responsibility for component binding to a DI *container* [30]. With DI, components only know each other's interfaces, and it is up to the DI container to create and set (or *inject*) into a (client) component an instance of another (provider) component implementing a required interface. These relationships are shown in Figure 2 (center). A DI container is a runtime entity that links client components to provider components.

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

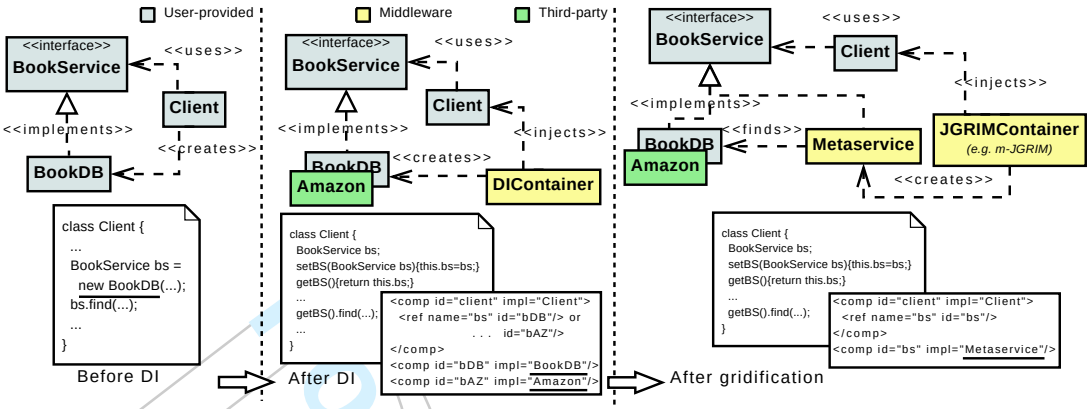


Figure 2: DI and metaservice injection

Consider an application that implements a book catalogue (BookService) and a Client component using it. The former is implemented via a relational database (BookDB). Figure 2 (left) depicts the relationships between these components. Roughly, Client setups an instance of BookBD by providing it with initialization parameters (database location, drivers, username/password). Though Client is only interested in browsing books (the operations of BookService), it has to know implementation details of BookDB, thus it is coupled both to the catalogue interface and its implementation.

Figure 2 (center) shows the DI version of the application. The DI container now injects a concrete implementation of BookService, such as a BookDB or a Web Service interface to Amazon Books. Consequently, DI removes the dependency between the client and the service implementation, since Client no longer instantiates BookDB. Besides, any implementation of BookService can be used without modifying Client. JGRIM takes DI a step further by introducing an indirection between components to inject Grid metaservices (right of Figure 2). After gridification the container no longer injects a service implementation into the client, but a metaservice, which is for example able to dynamically discover an implementation residing in a Grid. The client interacts with the metaservice, which in turn interacts with an instance of the required service. This indirection is transparent to the client, this is, there is no need to change its code, since both the service implementation and the metaservice realize BookService. Besides discovery and invocation, metaservices also represent tuning services. For example, the above metaservice could choose the fastest available book service instance.

Upon gridification, the developer must select which component dependencies should be enhanced with Grid capabilities. This involves to prepare his application so that a JGRIM container can then inject metaservices. To this end, JGRIM prescribes a *gridification process* (Figure 3) of four steps:

1. *Hot-spot identification*: A developer identifies the dependencies (*hot-spots*) into which meta-services are injected. In the figure, hot-spots have been sketched with dashed lines. For example, the implementation of C may be outsourced to a third-party service. Then, the dependency from B to C may be equipped with runtime service discovery. By drawing a parallel with

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

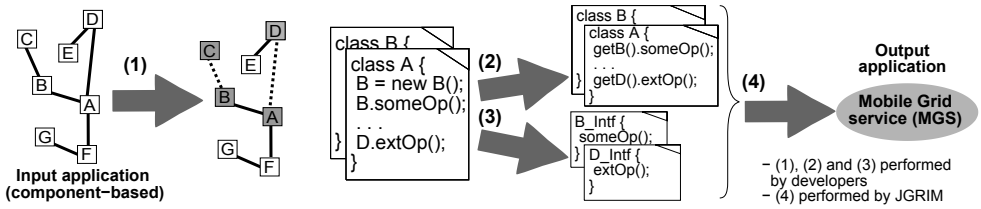


Figure 3: JGRIM: Gridification process

the above example, B and C would be the Client and BookDB components, respectively. The dependency A-D may be enhanced, for instance, with fault tolerance.

Components like C, for which the application does not provide an implementation, are called *external*. Conversely, components like A, B and D are called *internal*. A dependency involving two internal components is an internal dependency. An external dependency (e.g. B-C) originates when an internal component accesses an external one. JGRIM also defines a *self* dependency, which is the case when the two components of the dependency are the same. The next section will exemplify these types of dependencies and the metaservices that can be injected into them.

2. *Component interface definition*: Here, the developer specifies the interfaces of the internal and external application components so as to separate what a component does from how it does it. For the internal interfaces, this is a common practice in Java, thus the task is seldom necessary [1]. For the external ones, it involves specifying the method signatures of the outsourced services.
3. *Coding conventions*: Involves modifying the application to ensure that its components follow the JavaBeans specification [31]. Any reference to a component Comp within the code must be done by calling a fictitious getComp(), instead of accessing it directly as Comp.operation(). For example, reading data from a file component should be done by invoking getFile().read() instead of file().read(). These conventions must be followed for both internal and external dependencies. As this coding style is commonplace in Java, this step often requires little effort [1].
4. *Assembly and deployment*: JGRIM combines the outputs of (2) and (3) and deploys the gridified application on a Grid. Under the current materialization of JGRIM, m-JGRIM, this application is a Mobile Grid service (MGS) capable of migrating based on environmental conditions such as CPU availability, network latency and bandwidth, etc. MGSs are described in Section 4.

JGRIM is a technology-agnostic gridification method, in the sense that the above process does not prescribe specific technologies either for implementing metaservices or associating them to component dependencies. This is precisely the role of m-JGRIM. The next subsection illustrates JGRIM concepts through the gridification of an application in the context of m-JGRIM.

3.2. An example: Image restoration

This section describes the gridification of an application for image restoration. Anatomically, the application follows the master-worker pattern. Target images are located at a remote repository (e.g.

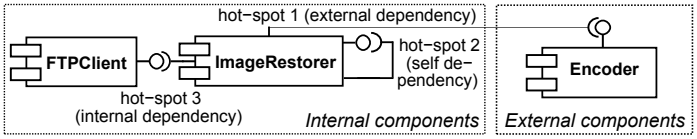


Figure 4: Components of the image restoration application

FTP). The master operates by downloading and splitting an image into two halves, which enables the application to exploit dual core CPUs. Then, the master assigns each subimage to a worker for restoration, joins the results, and encodes the joined image into a bitmap format.

Suppose we have already implemented some of the components of the application, including an ImageRestorer (for restoring images and subimages) and an FTPClient (for transferring files and obtaining file metadata). As we are not providing a component for image encoding, we will outsource an implementation from the Grid. ImageRestorer is the master that coordinates the whole restoration process. Enhancement of individual halves of an input image is handled by two concurrent workers (threads). After processing subimages, an encoder component is used to generate the final image:

```
public class ImageRestorer {
  public byte[] restoreImage(String imageURI, String format) {
    byte[][] halves = split((new FTPClient()).getFile(imageURI));
    WorkerThread worker0 = new WorkerThread(this, halves[0]);
    WorkerThread worker1 = new WorkerThread(this, halves[1]);
    worker0.start(); worker1.start();
    worker0.join(); worker1.join(); // Wait until child threads are finished
    byte[] result = combine(worker0.getResult(), worker1.getResult());
    return encoder.encode(result, format); // Access to a missing component
  }
  public byte[] restoreSubimage(byte[] imgData) {...}
}
public class FTPClient {
  public FileMetadata getMetadata(String fileURI) {...}
  public byte[] getFile(String fileURI) {...}
}
public class WorkerThread extends Thread {
  private ImageRestorer restorer = null;
  private byte[] myHalf = null, result = null;
  public WorkerThread(ImageRestorer restorer, byte[] myHalf) {
    this.restorer = restorer; this.myHalf = myHalf;
  }
  public void run() { result = restorer.restoreSubimage(myHalf); }
  public byte[] getResult() { return result; }
}
```

Figure 4 depicts the component diagram of the application. We will take this code and generate its gridified counterpart. For clarity reasons, we will not follow the process as presented in Section 3.1, but take one hot-spot at a time and incrementally carry out the subsequent steps (2) and (3) of the process.

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

3.2.1. *Hot-spot 1: The ImageRestorer-encoder dependency*

The first hot-spot is the ImageRestorer-encoder dependency. We have provided the expected interface at the client-side for the encoding service (ImageEncoderIF) and altered the code of ImageRestorer so that all accesses to encoder are performed through a getEncoder() method. Conceptually, JGRIM takes advantage of this code structure to inject a metaservice that dynamically finds a service adhering to that interface. Then, processing this information with m-JGRIM results in:

```
1  public class ImageRestorer extends core.MGS {
2    ImageEncoderIF encoder = null;
3    public ImageEncoderIF getEncoder() { return encoder; }
4    public void setEncoder(ImageEncoderIF encoder) { this.encoder = encoder; }
5    public byte[] restoreImage(String imageURI, String format) {
6      ...
7      return getEncoder().encode(result, format); // Interaction with an external component
8    }
9  }
10 public interface ImageEncoderIF {
11   public byte[] encode(byte[] imgData, String format);
12 }
```

m-JGRIM applications automatically inherit from the MGS class, which provides mobility primitives. Note that m-JGRIM added an instance variable (encoder) and getter/setters for accessing it (lines 2-4). These instructions enable m-JGRIM to non-invasively set service discovery and invocation capabilities to ImageRestorer through DI. Currently, these metaservices are implemented by m-JGRIM through runtime inspection of UDDI registries and invocation of WSDL-interfaced services (see Section 4.1). Moreover, metaservices are associated to the application through an automatically generated file:

```
<?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="mainComponent" class="ImageRestorer">
    <property name="encoder"><ref bean="encoderService"/></property>
  </bean>
  <bean id="encoderService" class="WSDLMatcherPortProxyFactoryBean">
    <property name="proxyInterfaces">ImageEncoderIF</property> ...
  </bean>
</beans>
```

The file links application components and metaservices together forming a fully operative application. Here, ImageRestorer is made dependent –via the encoder property– of an encoderService metaservice whose interface is ImageEncoderIF. The two benefits of this approach are that components are decoupled, since binding to external Grid services is performed at runtime, and encoderMetaService can be easily mocked for testing purposes. For performing DI, m-JGRIM is based on Spring [30], a DI library that also features support for Web Services and aspect-oriented constructs.

3.2.2. *Hot-spot 2: The ImageRestorer-ImageRestorer dependency*

As shown, restoreImage in ImageRestorer issues two asynchronous calls to restoreSubimage. These calls are independent between each other, hence they are executed concurrently by using threads. However, Grids offers many alternatives to threads to handle the execution of parallel computations

that come as sophisticated services. These services, besides parallelism, may offer scheduling, load balancing, fault tolerance, etc. JGRIM exploits such execution services through self dependencies.

Self dependencies originate when a component calls its own methods. Applying steps (2) and (3) of the gridification process to our analyzed hot-spot results in: (a) defining an interface including all those methods that are subject to concurrent execution, and (b) using getters when invoking such methods. Let us suppose we name this interface SubImageRestorerIF. Then, the gridified code is:

```
1  public class ImageRestorer extends core.MGS {
2    ...
3    SubImageRestorerIF self = null;
4    public SubImageRestorerIF getSelf() { return self; }
5    public void setSelf(SubImageRestorerIF self) { this.self = self; }
6    public byte[] restoreImage(String imageURI, String format) {
7      ...
8      byte[] result0 = getSelf().restoreSubimage(halves[0]);
9      byte[] result1 = getSelf().restoreSubimage(halves[1]);
10     byte[] result = combine(result0, result1); // Wait until computations are finished
11     ...
12   }
13 }
14 public interface SubImageRestorerIF {
15   public byte[] restoreSubimage(byte[] imgData);
16 }
```

To Grid-enable our self dependency, we must replace the asynchronous calls to restoreSubimage by sequential calls to the same operation on self (lines 8-9). Then, m-JGRIM appends the code to support DI for this component (lines 3-5). The only extra programming convention needed for the mechanism to work is that the results of the parallel computations must be placed on local variables (lines 8-9). Further references to these variables (e.g. line 10) will transparently block the execution of restoreImage until they are computed by a special middleware component that intercepts and executes both calls concurrently. Behind scenes, m-JGRIM further preprocess the code of calling methods (e.g. restoreImage) to add instructions for synchronization purposes through the use of Java futures. To configure the self dependency, a new metaservice is added to the above XML file:

```
<beans>
  <bean id="mainComponent" class="ImageRestorer">
    ... <property name="self"><ref bean="executionService"/></property>
  </bean>
  <bean id="executionService" class="CondorBasedExecutionProxyFactoryBean">
    <property name="proxyInterfaces">SubImageRestorerIF<property/> ...
  </bean>
</beans>
```

which adds to the application –via the self property– a concrete implementation of a service for concurrently executing the operations defined in SubImageRestorerIF. Here, we submit such operations to Condor. Interestingly, the logic is free from (threading) parallelization code. More important, execution of spawned methods can be seamlessly handled by means of execution mechanisms suitable for exploiting Grids. Section 4.3 discusses the support of m-JGRIM to leverage such services.

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

3.2.3. *Hot-spot 3: The ImageRestorer-FTPClient dependency*

The last hot-spot for gridification in our application is the ImageRestorer-FTPClient dependency. Again, we have to isolate the implementation of FTPClient behind an interface (e.g. FileDownloader). Note that this practice improves flexibility and extensibility. For instance, it is now easier to use another component for transferring files (e.g. GridFTP [32]), as long as this component adheres to FileDownloader. Moreover, direct usage of FTPClient is disallowed, thus we have to replace any access to FTPClient within the code by calls to the corresponding getter (e.g. getDownloader()).

m-JGRIM applications are mobile Grid entities. Indeed, mobility can bring benefits in terms of decreased latency and bandwidth usage when applications are moved to locally interact with remote data. Particularly, an interesting performance improvement is to move the application to the repository location when the size of the target image file exceeds some threshold, this is, we could customize the interaction between ImageRestorer and FTPClient. In m-JGRIM, this kind of tweaks are introduced through *policies*, which are special components that mediate between the two elements of a dependency. Then, mobility can be added to our hot-spot by attaching a policy to it:

```
1  public class MovePolicy extends core.policy.PolicyAdapter {
2    public void executeBefore() {
3      // Obtains the file to download from the execution context of transferFile(imageURI)
4      String fileURI = (String)getExecContext().getOperationArgument(0);
5      FileDownloader downloader = (FileDownloader)getExecContext().getTargetComponent();
6      if (downloader.getMetadata(fileURI).getSize() > 524288) {
7        MGS app = (MGS)getExecContext().getSourceComponent();
8        app.moveTo(parseServerLocation(fileURI));
9      }
10   }
11 }
```

Upon each interaction between ImageRestorer and FileDownloader, MovePolicy is evaluated. The code within executeBefore is executed just before the invocation of an operation defined in FileDownloader takes place. Analogously, an executeAfter method can be specified. The metainformation about the operation being executed is made accessible to programmers through the getExecContext method from the policy framework (lines 4, 5, and 7). Concretely, the policy moves the application to the node hosting the data (line 8) if the size of the image file exceeds 512 KB. In such a case, the "downloading" process will be started locally at that host. Policies can be configured to act upon invocations on specific operations of the target component. Here, we want MovePolicy to be activated only when transferFile is called. Section 4.4 explains the configuration generated to support the injection of policies.

Policies can be also employed to customize external dependencies. For example, let us suppose that the restoration application is deployed on a Grid where many nodes host an instance of the encoder service. Additionally, let us assume that bandwidth across nodes could drastically vary along time. Under these conditions, accessing a service replica through a busy network link might compromise the application response time. We can control which instance is chosen for serving each call to encode(imageData, format) by attaching a policy to the ImageRestorer-encoder dependency:

```
public class BandwidthPolicy extends core.policy.ExternalPolicyAdapter {
  public String accessFrom(String wsdlURI_1, String wsdlURI_2) {
    double bw1 = core.policy.Profiler.instance().getBandwidth("localhost", wsdlURI_1);
    double bw2 = core.policy.Profiler.instance().getBandwidth("localhost", wsdlURI_2);
    return (bw1 < bw2) ? wsdlURI_1 : wsdlURI_2;
  }
```

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

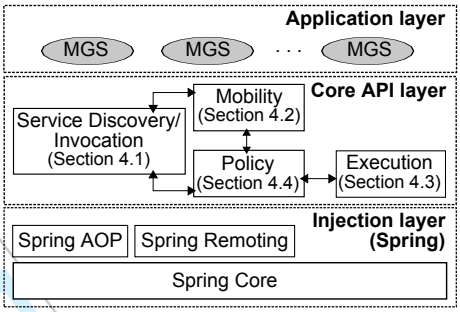


Figure 5: Architecture of m-JGRIM

}

Roughly, accessFrom is a hook by which developers can specify the rules that govern Web Service selection. BandwidthPolicy tells the application to use the service instance hosted at the Grid node to which the host where the MGS is executing experience the best bandwidth.

4. m-JGRIM: A PROOF-OF-CONCEPT MATERIALIZATION OF JGRIM

As explained, JGRIM is essentially based on transparently injecting metaservices, which are entities provided at the middleware level that provide Grid behavior to ordinary applications. While this practice has many advantages from an engineering perspective [1], achieving such a transparency is indeed a daunting task from a technological standpoint. In this section we describe m-JGRIM, a novel Grid middleware that materializes JGRIM and as such bridges this gap.

Figure 5 shows the architecture of m-JGRIM, which comprises three layers: *Application* (represents gridified applications), *Core API* (provides access to concrete Grid services through components that materialize metaservices), and *Injection* (seamlessly wires application components and metaservices together through a DI container). After passing through the m-JGRIM gridification process, an ordinary application becomes a mobile entity called MGS, which can move across the nodes of a Grid to locally access resources. MGSs are created by injecting metaservices into the corresponding non-gridified code, which are supplied at the Core API layer. Metaservices are implemented through middleware-level components that either wrap existing Grid services (e.g. UDDI discovery) or materialize new ones (e.g. mobility, policies). These components are grouped in four subsystems:

- *Service Discovery/Invocation subsystem* (Section 4.1): Performs Grid service discovery and invocation by providing concrete bindings between ordinary applications and Grid services. Currently, service discovery is supported by inspection of UDDI registries, while service invocation is performed by extending the remoting facilities of Spring.
- *Mobility subsystem* (Section 4.2): Offers migration capabilities to gridified applications. m-JGRIM features both explicit and implicit strong mobility [33].

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

- *Execution subsystem* (Section 4.3): Provides support for associating concrete Grid execution services to self dependencies, thus leveraging existing services for parallelism, load balancing and fault tolerance. At present, m-JGRIM is integrated with Satin [20], a module of Ibis [15].
- *Policy subsystem* (Section 4.4): Is an extensible framework that allows developers to specify decisions related to application tuning regarding both internal and external dependencies.

The Injection layer relies on Spring [30], a DI framework including several built-in modules with common middleware functionality. Entities instantiated by Spring are components that follow the JavaBeans specification called *beans*. There are many DI frameworks for Java[†], but we use Spring since it is widely popular among Java developers. The rest of the section describes the subsystems of m-JGRIM. From now on, by "bean" we will refer to components supplied at the Core API or Injection layers. Similarly, by "application bean" we will refer to application-specific components.

4.1. The Service Discovery/Invocation subsystem

The Remoting module of Spring provides convenient programming abstractions for working with various RPC technologies. The module isolates applications from the intricate configuration and coding details involved in calling remote services. This separation is achieved by proxying such services with special beans that decouple client applications from the protocols to access remote services.

The JaxRpcPortClientInterceptor bean provides access to Web Service operations via JAX-RPC, a specification for interacting with WSDL-interfaced Web Services. Application beans can define a dependency to a Web Service by supplying the dependency interface and the information for contacting the service. For example, the following code shows the Spring configuration for an application bean declaring a dependency to a currency converter Web Service:

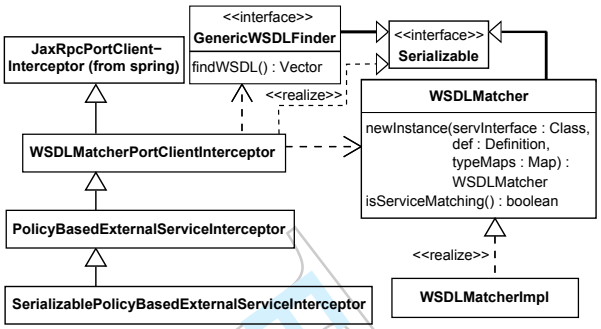
```
1  <beans>
2    <bean id="client" class="example.CurrencyConverterApp">
3      <property name="currencyService"><ref bean="currencyWebService"/></property></bean>
4    <bean id="currencyWebService" class="org.springframework...JaxRpcPortProxyFactoryBean">
5      <property name="portInterface">example.ICurrencyConverterService</property>
6      <property name="wsdlDocumentUrl">http://example.edu/currency?WSDL</property>
7      <property name="namespaceUri">http://example.edu/currency</property>
8      <property name="serviceName">CurrencyService</property>
9      <property name="portName">CurrencyPort</property></bean>
10 </beans>
```

Lines 6-9 are the contact information of the Web Service, and *portInterface* is the service contract to which client must adhere. At runtime, Spring creates a proxy and injects it into client to transparently translate any method call issued on the dependency interface (ICurrencyConverterService) to the corresponding operation of the Web Service (currencyWebService). In this way, Spring follows a *contract-first* approach to service consumption: the client-side interface specified for a required service must exactly match the interface of the Web Service at the server-side. The developer has therefore to know *in advance* the interface of any external service before using this latter in his application.

Conversely, m-JGRIM provides a specialized bean that mediates between the interface of an external dependency and the actual interface of a Web Service. The bean gets rid of the configuration

[†]A comprehensive list can be found at http://en.wikipedia.org/wiki/Dependency_injection#Existing_frameworks

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961



(a) Simplified class diagram

```
procedure MATCHES(interface,WSDL)
  wsdlInterface ← WSDL2JAVA(WSDL)
  for all method ∈ interface do
    isSupported ← false
    for all wsdlMethod ∈ wsdlInterface and
          isSupported = false do
      isSupported ← MATCHES(method,wsdlMethod)
    end for
    if isSupported = false then
      return false
    end if
  end for
  return true
end procedure
```

(b) The matches(interface,WSDL) primitive

Figure 6: The Service Discovery/Invocation subsystem

details for contacting services (i.e. WSDL location, namespace, etc.) and otherwise extracts the information from Web Service registries. Figure 6 (a) shows a simplified class diagram of this support. This bean is called a *service discovery bean* (SDB) and is implemented by extending the Web Service support of Spring. GenericWSDLFinder represents registries of WSDL descriptions. Currently, discovery in m-JGRIM is based on UDDI, but more discovery protocols can be added by realizing GenericWSDLFinder (e.g. WS-QBE [34]). Moreover, WSDLMatcher determines whether an individual WSDL contains the method signatures of a dependency interface like ICurrencyConverterService, this is, it models a matches(interface, WSDL) primitive. The default implementation for this primitive (WSDLMatcherImpl) is shown in Figure 6 (b).

The algorithm converts the input WSDL to a Java interface (wsdlInterface), and then checks whether the operations specified in interface are included in wsdlInterface. The matches(method, wsdlMethod) function matches two method signatures. Two methods match if they have the same name, the same number of arguments, and a one-to-one correspondence between argument and return types can be established. Immutable Java types are subject to an exact match. Matching of object types can be customized by specifying mappings between client-side and server-side types. Lastly, array-based types match if their associated basic types also match according to the previous rules.

PolicyBasedExternalServiceInterceptor implements a *policy-based SDB* (PSDB), which contacts Web Services based on policies (see Section 4.4). After querying a UDDI registry, a PSDB passes on the candidate Web services to its associated policy bean(s) to find out which service instance must be used, and how it must be contacted. Consequently, a PSDB bean may, for example, remotely invoke the service or trigger the migration of the application to the node where the service is hosted instead. SerializablePolicyBasedExternalServiceInterceptor provides support for using PSDBs in conjunction with mobility beans (see Section 4.2).

To illustrate the DI-related configuration generated to use plain SDBs, let us inject service discovery (without policies) into the application discussed at the beginning of this section. Instead of supplying the application a hardcoded reference to a currency converter service, we will inject an SDB that dynamically discovers a Web Service implementing ICurrencyConverterService:

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

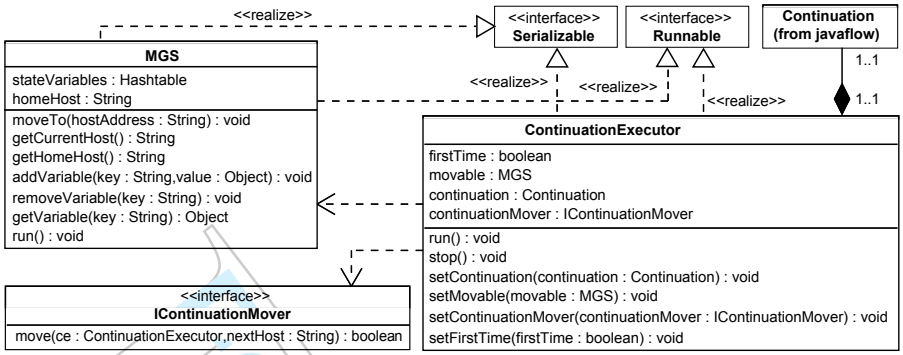


Figure 7: Class design of the Mobility subsystem

```
1  <beans>
2    <bean id="client" class="example.CurrencyConverterApp">
3      <property name="currencyService"><ref bean="currencyWebService"/></property>
4    </bean>
5    <bean id="currencyWebService" class="WSDLMatcherPortProxyFactoryBean">
6      <property name="proxyInterfaces">example.ICurrencyConverterService</property>
7      <property name="wsdlMatcher"><ref bean="wsdlMatcher"/></property>
8      <property name="wsdlFinder"><ref bean="wsdlFinder"/></property>
9    </bean>
10   <bean id="wsdlMatcher" class="WSDLMatcherImpl"/>
11   <bean id="wsdlFinder" class="UDDIFinder"><!-- UDDI-specific parameters --></bean>
12 </beans>
```

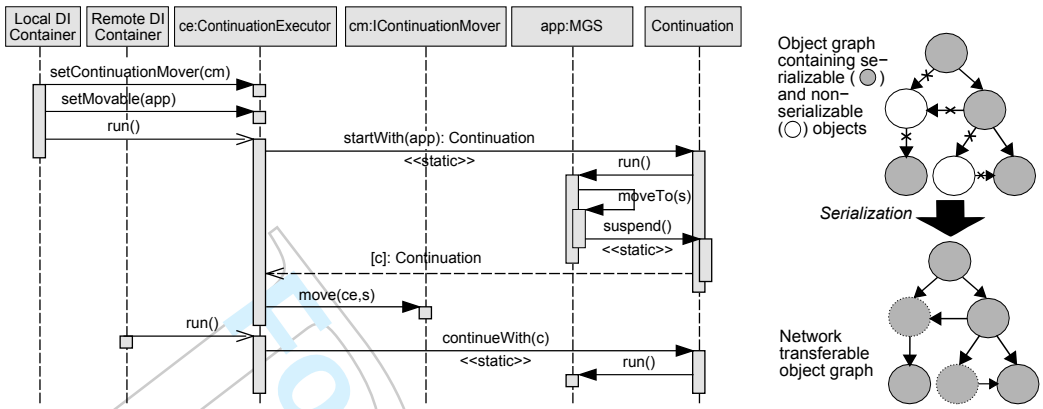
The client now accesses the currency service through an SDB (line 5). Web Service matching is implemented by the wsdlMatcher bean (line 10), while UDDI inspection is performed by the wsdlFinder bean (line 11), which holds the location and the authentication information of the UDDI registry.

4.2. The Mobility subsystem

Applications gridified with m-JGRIM automatically extend the MGS core class, which provides basic primitives for handling mobility and managing application-specific state. These primitives are intended to be invoked from within policies so as to keep the application logic clean from the m-JGRIM API. We call this *implicit* mobility. Nevertheless, developers can also use mobility in an *explicit* way, this is, from within the application logic. The class design of this subsystem is shown in Figure 7.

m-JGRIM implements a *strong* migration mechanism [33]. *Strong* refers to the ability of a runtime system to support migration of both the binary code and the execution state of a running application. When an application migrates from a host H_1 to a host H_2, its execution is resumed at H_2 from the point it left off when executing at H_1. In opposition, *weak* migration [33] cannot transfer the execution state of applications. Developers must programmatically save and restore the execution state of their

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961



(a) Initiating, suspending and resuming MGS execution: sequence diagram. The call to moveTo(s) can be either implicit or explicit

(b) Serialization in m-JGRIM

Figure 8: Strong migration in m-JGRIM

applications, which has a negative impact in application design and implementation. Ibis is another middleware featuring strong migration, while ProActive and Babylon rely on weak mobility.

The MGS class mostly implements mobility-related functionality. The methods getCurrentHost and getHomeHost return the IP of the hosts where an application is currently executing and where it was initiated, respectively. Moreover, moveTo migrates the application to a host by capturing and storing the execution of the MGS into a *continuation*, which contains a snapshot of the stack trace, local variables and program counter. This information is used to restore the execution of the suspended application once it has been migrated to a remote host. Support for continuations is based on Javaflow [35], a library to fully capture the execution state of Java threads that internally relies on bytecode instrumentation techniques. Each instance of MGS is injected a ContinuationExecutor bean, which actually controls the transference and restoration of Continuation objects. This bean is in turn injected an IContinuationMover bean, which represent transport mechanisms for transferring continuations (e.g. sockets, RMI, etc.). Figure 8 (a) depicts the process of executing and migrating an MGS.

Besides moving execution state, m-JGRIM also transfers the bytecode of an application when this code is not present at the destination host. When a host receives a continuation, m-JGRIM transfers from the origin host the missing Java classes to fully restore the execution of the application by using a special network classloader. Received classes are stored on disk thus they can be sent to other hosts too. In consequence, the deployment of MGS code across a Grid is done incrementally and without involving the application developer.

A problem that arose when combining mobility with Spring concerned serialization. Java objects can be transferred through a network provided they are either Serializable or Externalizable. Since for DI purposes Spring make extensive use of classes which were not thought to be transferable (e.g. dynamic proxy classes), marshalling the execution state of MGSs via standard Java serialization is unfeasible. We designed a serialization mechanism for converting objects into a serializable form, which works

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961


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

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

Figure 9: Self dependencies in m-JGRIM

by modifying at runtime the bytecode of an object to force it to be serializable by using ASM [36]. This transformation is also recursively done on the object's attributes to ensure that its whole object graph is serializable (see Figure 8 (b)). Another problem concerns Java single inheritance, which may prevent codes already using extension from being gridified with m-JGRIM. Nevertheless, this issue can be addressed by using existing techniques that rely on object wrapping and Java reflection [37].

4.3. The Execution subsystem

The execution metaservices of m-JGRIM rely on Spring *interceptors* [30], beans that transparently introduce behavior before/after certain application methods are invoked. Spring features an extensible support for developing interceptors, and offers built-in interceptors to add cross-cutting concerns (logging, profiling, debugging, etc.) into applications without code modification.

Commonly, interceptors are attached to application dependencies. If an application bean *A* depends on another application bean *B*, an interceptor can act upon method calls from *A* to *B* to transparently do something in between. Recall that Spring interceptors were also used to support external dependencies (Section 4.1), where *A* and *B* are an application bean and a Web Service, respectively, and the interceptor is an SDB or a PSDB. As for self dependencies, *A* is the same application bean as *B*. In this context, interceptors are the beans that provide concurrency for self dependency methods.

Let us revisit the usage of self dependencies in JGRIM. Basically, they are specified by defining an interface including the methods whose execution is delegated to specialized execution services. Access to these methods in the application code is done by using a getter instead of calling them directly. For example, upon gridification, the ordinary code of Figure 9 (left) must be modified by the developer so as to access a fictitious self application bean, and to store the result of methodB in a local variable. In addition, the interface of self must be defined. These tasks produce the code of Figure 9 (right). m-JGRIM then makes the modified code to inherit from the MGS class, and adds an instance variable of type ParallelIF and its getter/setter. The calls to methods in ParallelIF (i.e. methodB) are executed concurrently with the invocation to the calling method (i.e. methodA). Besides, m-JGRIM modifies the body of calling methods to insert barriers that block their execution until the results of concurrent computations are available. To this end, m-JGRIM relies on the java.util.concurrent package of Java.

At runtime, the execution of self dependency methods is handled by an *execution bean* (EB), whose definition is appended to the XML configuration of the application being gridified. EBs intercept and

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

forward any call to such methods to existing Grid resource management systems. Therefore, EBs know the protocol(s) to talk to the execution service of a particular middleware. Returning to the example, and assuming Satin as the target system, the generated configuration is:

```
1  <beans>
2    <bean id="app" class="example.MyApplication">
3      <property name="self"><ref bean="self"/></property></bean>
4    <bean id="self" class="org.springframework.aop.framework.ProxyFactoryBean">
5      <property name="proxyInterfaces">example.ParallelIF</property>
6      <property name="interceptorNames"><list><value>executor</value></list></property>
7    </bean>
8    <bean id="executor" class="SatinInterceptor">
9      <property name="ownerApp"><ref bean="app"/></property></bean>
10 </beans>
```

The main application bean (*app*) declares a dependency to a *self* application bean (line 3), supported via a factory bean (line 4) that instantiates the *executor* EB. The calls to the methods of self (i.e. those defined in ParallelIF) are intercepted by *executor* (line 6), which delegates the execution of those methods to Satin (line 9). Currently, m-JGRIM offers thread-based and Satin-based EBs. The former operates by running each method call in a separate thread from a shared pool. The latter allow applications to exploit the services of Satin/Ibis for executing divide and conquer methods on Grids. Lastly, the development of EBs for using the execution services of ProActive and Condor is underway.

4.3.1. The Satin EB

Satin EBs handle the execution of divide and conquer self dependency operations, and to exploit the parallelism and load balancing capabilities of Satin/Ibis but without modifying the application code to use the Satin API. The only requirement imposed to the developer for using Satin EBs is to place the results of recursive calls on local variables. This is a simple modification that does not involve using Grid APIs within the application code.

When building pure Satin applications, developers must obey some code conventions, namely subclassing an API class and declaring an interface with the methods whose execution is spawned. Developers must also include synchronization barriers. m-JGRIM automates these tasks. Injecting a Satin EB into an MGS triggers the creation of a *peer* whose code is automatically derived from the MGS but altered so it follows these conventions. At runtime, the peer is indirectly used by the MGS through a Satin EB. Recall the example application at the beginning of this section, which declared a self dependency on a methodB operation. Let us suppose methodB is a CPU-intensive recursive algorithm, thus it may be run with Satin. The peer created by m-JGRIM is:

```
public interface ParallelIF_Peer extends ibis.satin.Spawnable {
  public boolean methodB();
}
public class MyApplication_Peer extends ibis.satin.SatinObject implements ParallelIF_Peer {
  // Variables and dependencies of the owner MGS are copied here
  public boolean methodB() {
    boolean aBranch = methodB(); // spawned by Satin
    boolean anotherBranch = methodB(); // also spawned
    ...
    super.sync(); // Satin barrier (automatically inserted)
    return (aBranch || anotherBranch);
  }
```

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

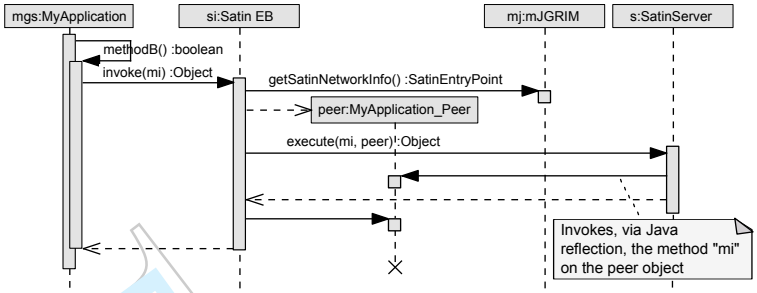


Figure 10: Execution of self dependency methods under Satin

}

Moreover, since it is composed of spawnable calls to itself, methodB is analyzed and modified by m-JGRIM to include calls to the sync Satin barrier, which ensures that the results of recursive calls are available before they are read. Automatically inserting sync simplifies programming and isolates developers from the Satin API, thus gridification is easier. Besides, programmers who are familiarized enough with Satin can modify the generated peer to introduce well-known optimizations in divide and conquer programs such as using a threshold on the number of spawns.

Upon execution of getself().methodB(), the associated Satin EB instantiates and sends the peer for execution to a Satin (see Figure 10). Particularly, a call to methodB causes the EB to create an instance of MyApplication_Peer by setting to this latter the MGS instance variables/dependencies, which is necessary since the original recursive methods may use this data. The Satin server is an extended Satin runtime that allows peers to be submitted to Satin in a client-server fashion. Eventually, the computation finishes and the server delivers the result back to the EB, which in turn passes it to the MGS. More implementation details on this mechanisms can be found in [38, 39].

4.4. The Policy subsystem

Policies [1, 29] offer a non-invasive, programmatic support to tune m-JGRIM applications. To code policies, programmers only have to learn a small subset of the m-JGRIM API. In addition, the separation between the tasks of implementing the logic of an application and associating policies to it brings benefits to the process of gridifying an application itself, since these tasks can be performed independently by developers with different skills on Grid programming.

The class design of the Policy subsystem is shown in Figure 11. Policy represents a generic bean that can act before and after a method of some dependency is invoked. Methods executeBefore and executeAfter are hooks to specify custom actions that are executed upon a call to any method of the dependency to which the policy is associated. Policies can be temporarily activated/deactivated by specializing isActivated. Lastly, all policies have a reference to metadata information about the operation being executed. In this way, policies are granted access to the state and behavior of the

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

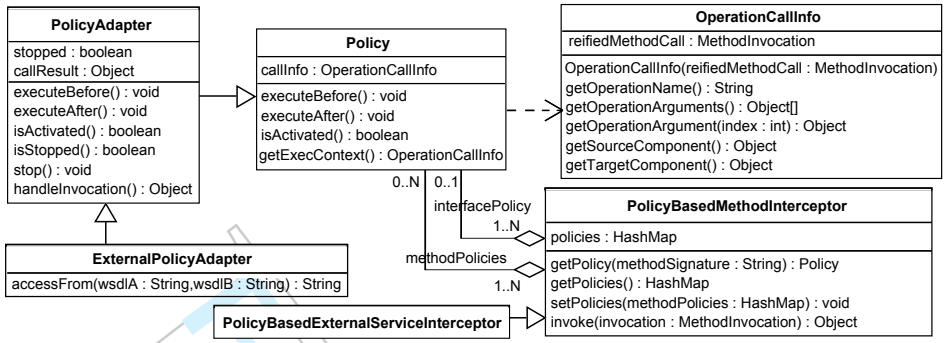


Figure 11: Class design of the Policy subsystem

application beans they control. PolicyAdapter extends Policy with methods to permanently cease the activity of a policy (stop) and to programmatically invoke a dependency method (handleInvocation). For example, one might implement a policy to cache the results of the calls to methods of internal/external dependencies by coding handleInvocation thus the results of some invocations are extracted from a cache maintained by the policy. Not overriding handleInvocation means that m-JGRIM will carry out the call directly on the target application bean. Moreover, ExternalPolicyAdapter models a full-fledged m-JGRIM policy that customizes the interaction with external Web Services.

Instances of PolicyAdapter are used by the PolicyBasedMethodInterceptor bean, which represents policy-enabled internal dependencies. The *policies* attribute is a map containing policies that act upon invocations on methods of the dependency interface. Each entry of the map is a (key, policy) pair, where *key* is a regular expression that represents the method signatures controlled by *policy*. Below is the XML configuration generated for an application that uses a policy between the interaction of two internal application beans *A* and *B*, this latter with interface B_Intf:

```
1  <beans>
2    <bean id="A" class="...">
3      <property name="B"><ref bean="policyHandler"/></property></bean>
4    <bean id="policyHandler" class="org.springframework.aop.framework.ProxyFactoryBean">
5      <property name="proxyInterfaces">B_Intf</property>
6      <property name="interceptorNames">
7        <list><value>policyInterceptor</value>
8              <value>B</value></list>
9      </property>
10   </bean>
11   <bean id="policyInterceptor" class="PolicyBasedMethodInterceptor">
12     <property name="policies">
13       <map><entry><key>m*</key><ref bean="somePolicy"/></entry></map>
14     </property>
15   </bean>
16 </beans>
```

m-JGRIM injects a policyHandler bean into *A* (line 3). This bean, which is implemented through the AOP support of Spring, intercepts the calls to methods defined in B_Intf (line 5) and delegates

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961

their execution first to a policy (line 7) and, if not already handled via handleInvocation, to the target application bean (line 8). The methods of B_Intf which are subject to interception are listed in the policies property (line 12). Here, we considered the methods whose name starts with "m" (line 13).

The basic elements upon which policies are built are system metrics. m-JGRIM provides a well-defined Grid profiling interface on top of which tuning heuristics can be implemented, which currently supports the following metrics:

- $load(H)$: computes a forecast of the average CPU utilization (percentage) at the host H. For example, if an expensive operation of an internal application bean needs to be started, and the local CPU load is twice the load of a remote host R, the application may be moved to R. Similarly, $memLoad(H)$ obtains statistics about memory usage.
- $size(App)$: Returns the estimated size (in bytes) of the allocated memory for the object graph of an executing MGS. The metric is useful, when used in conjunction with network related metrics, to determine whether it is convenient to migrate an MGS.
- $latency(H_1, H_2)$: Represents the estimated latency (in seconds) for transmitting data between two hosts. This metric is crucial to decide, for example, which Web Service to contact from a set of available candidates. The profiling API also provides a primitive to estimate the network transfer rate (in KB/s), and the percentage of information lost during transference per unit time.

To return accurate values for the above metrics, m-JGRIM implements a distributed monitoring service that predicts the performance of both network and computational resources by employing regression models. Within a single Grid node, metrics are gathered by using JMX [40], while predicted values are communicated via GMAC [41], a P2P protocol that provides efficient multicast services across WANs.

5. EXPERIMENTAL EVALUATION OF M-JGRIM

This section describes the experiments that were carried out to provide evidence about the practical soundness of m-JGRIM. Section 5.1 focus on its evaluation at the macro level by comparing m-JGRIM with similar initiatives for gridifying applications with respect to code quality and execution efficiency. Section 5.2 describes microbenchmarks designed to measure the overhead of injecting m-JGRIM metaservices in terms of aspects such as time, memory and bytecode penalty.

5.1. Analysis of performance and network usage

We conducted a comparison between Ibis, ProActive and m-JGRIM by using these tools to gridify the k-NN algorithm (a popular data mining algorithm that employs a relational dataset to perform instance classification) and the picture enhancement application discussed in Section 3.2. The original codes were implemented by an experienced Java developer, whereas gridification was performed by another programmer with good skills in distributed Java development but minimal background on these middlewares. The execution of the parallel portions of the m-JGRIM applications were handled with Satin execution beans (Section 4.3). Basically, we decided to compare m-JGRIM against Ibis and ProActive as these tools have goals similar to our work, this is, to facilitate the construction of Grid applications while minimizing the need for API code upon gridification.

Table I. The experimental testbed (machines had single core CPUs)

Cluster	Internet bandwidth (Kbps)	Machine	CPU frequency (Ghz.)	Memory (MB)
C_1	2.048	$M_{1,1}$	0.83	256
		$M_{1,2}$	0.83	256
		$M_{1,3}$	0.83	384
		$M_{1,4}$	0.83	384
		$M_{1,5}$	0.78	256
		$M_{1,6}$	2.80	512
C_2	256	$M_{2,1}$	1.75	256
		$M_{2,2}$	1.83	1.024
C_3	256	$M_{3,1}$	2.00	1.024
		$M_{3,2}$	1.90	512

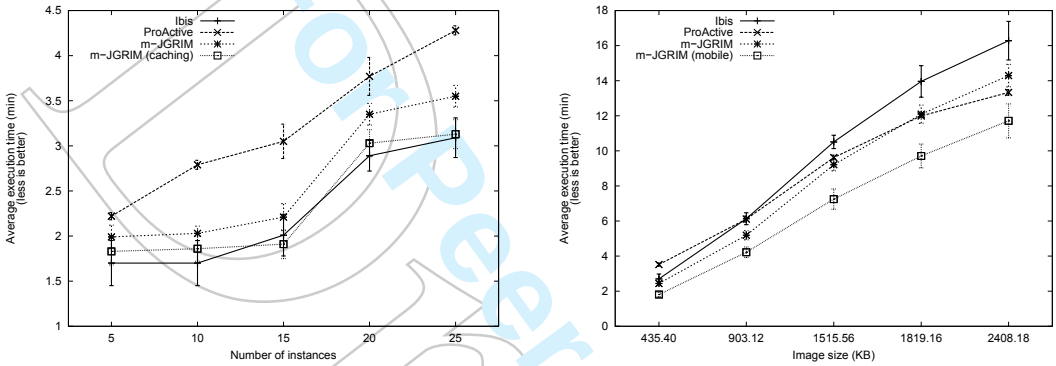
Experiments were performed on a Grid comprising three Internet-connected local clusters, each hosting a Web Service-interfaced replica of the dataset. Input images were stored on cluster C_1 and were accessible through FTP. All runs were launched from cluster C_2. Moreover, the experiments were performed during nighttime to avoid high Internet traffic and jitter (inter-cluster latency was 100-150 milliseconds). Table I details the characteristics of the machines of our experimental testbed.

We assessed the impact of gridification on the application code by comparing TLOC (total lines of code without considering neither blank nor comment lines) and GLOC (developer-supplied lines using Grid APIs/protocols for accessing Grid resources) metrics for the original codes and their gridified counterparts. Before measuring, all source codes were formatted with Eclipse. Table II summarizes these metrics (lower values are better). We did not take into account the configuration files in each case since m-JGRIM, unlike Ibis and ProActive, generates these files automatically. For the m-JGRIM applications we obtained two variants by implementing a caching policy for k-NN that keeps in main memory some dataset accesses to reduce network traffic, and a mobility policy for the image application that always moves the application to the FTP repository location to reduce network latency.

From the table it can be seen that, for both applications, m-JGRIM obtained good TLOC and GLOC. Ibis k-NN resulted in high TLOC as it does not fully support Web Services. Therefore, a lot of code had to be manually provided to interact with the dataset replicas. On the other hand, ProActive support for Web Service protocols is minimal. This feature, however, is necessary to achieve interoperability of Grid applications [9, 10]. Conversely, SDBs allowed m-JGRIM k-NN to smoothly delegate dataset discovery and access to the underlying middleware. Moreover, achieving parallelism with Ibis and ProActive demanded more API code. Remarkably, unlike its competitors, the m-JGRIM API was only used for coding policies, without affecting the original codes. This enforces the fact that using m-JGRIM may lead to more maintainable Grid code, since it follows a two-step approach to gridification in which the application logic is effectively isolated from the Grid-related code [1].

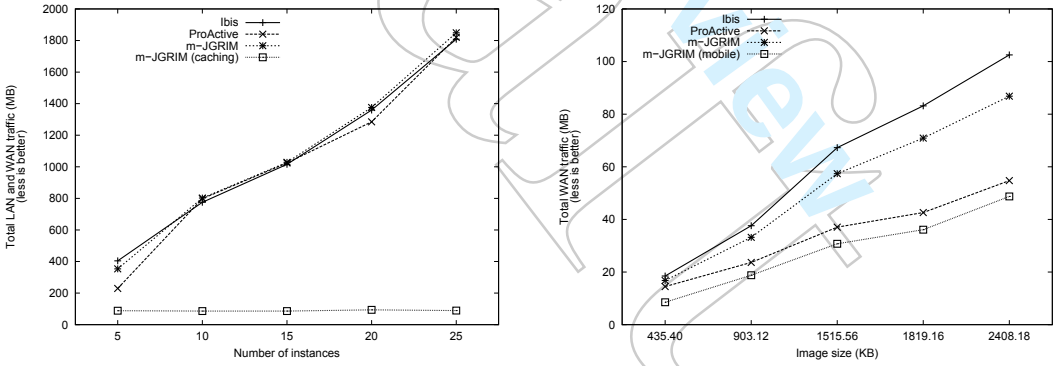
Table II. Test applications: code metrics

k-NN variant	TLOC	GLOC	Image app. variant	TLOC	GLOC
Original	192	–	Original	241	–
Ibis	1477	10	Ibis	227	5
ProActive	404	11	ProActive	299	17
m-JGRIM	**166**	**4**	m-JGRIM	**226**	**0**
m-JGRIM (caching)	179	6	m-JGRIM (mobility)	233	1



(a) k-NN (b) Image restoration

Figure 12: Execution time

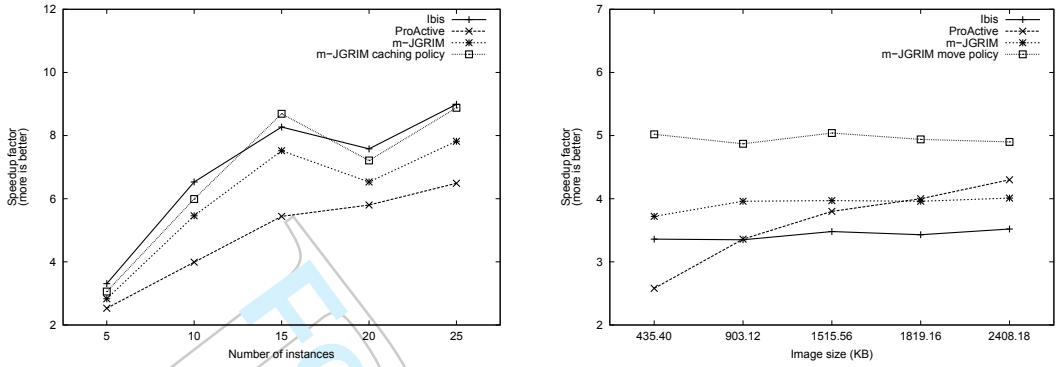


(a) k-NN (b) Image restoration

Figure 13: Network traffic

Prepared using speauth.cls

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961



(a) k-NN (b) Image restoration

Figure 14: Speedup

To evaluate runtime aspects, each gridified version of k-NN was used to classify several list of input instances with different sizes. For the image application we used five picture sizes. We averaged the execution time (AET) and accumulated the network traffic for 10 executions per test. AET deviations were around 5%. Loopback network traffic was filtered out. Figures 12 and 13 show the obtained results. As expected, m-JGRIM behaved similar to the alternatives. Besides, m-JGRIM policies (caching and mobility) improved both performance and network usage.

When not using the caching policy, m-JGRIM k-NN added a performance overhead of 10-15% compared to Ibis k-NN. However, this overhead was associated to service discovery, a desirable Grid feature not present in Ibis and limitedly supported in ProActive. Besides, caching allowed m-JGRIM to continue using service discovery and to stay competitive. ProActive k-NN, on the other hand, performed poorly. ProActive is oriented towards easy deployment, which makes application setup slower. This suggests that ProActive may not be suitable for moderately long running computations in which execution time is slightly greater than setup time. Moreover, caching significantly reduced network traffic, which is a consequence of performing less remote dataset accesses. Of course, Ibis and ProActive k-NN might have benefited from this caching technique, but this would have required yet more modifications to the original k-NN code as these middlewares do not follow a non-intrusive approach to application tuning. In other words, most performance improvements must be explicitly introduced in the application code, jeopardizing the maintainability of the resulting Grid application.

With respect to the image processing application, m-JGRIM (without the mobility policy) performed better than Ibis, due to the fact that m-JGRIM extends the scheduler of its Satin module with a remote client server-like job submission interface. In this way, Ibis and m-JGRIM applications are subject to different execution conditions. Moreover, ProActive generated the least amount of WAN traffic. Unlike Ibis and therefore m-JGRIM, its job scheduling algorithm is not subject to random factors. Nevertheless, mobility allowed m-JGRIM to improve performance and reduce this traffic. Again, the policy did not affect the original source code. Unfortunately, Ibis does not let developers to explicitly control mobility, whereas ProActive only offers weak mobility, which –as discussed in Section 4.2–

requires extensive code modifications to save and restore the execution state of computations across migrations.

Figure 14 shows the speedup achieved by the applications, computed as AET over the average execution time of the original codes on a single machine. In both graphics, the speedup curves of Ibis and m-JGRIM seemed to have the same behavior, as m-JGRIM also relies on Satin for parallelism. In addition, the random nature of the task scheduler of Satin (and hence our Satin EBs) caused Ibis k-NN and m-JGRIM k-NN to have lower speedups for larger experiments (see for example the dip in the curves associated to Ibis and m-JGRIM in Figure 14 (a) for 20 instances). To a lesser extent, this effect was also present in the image application. Furthermore, ProActive appeared to linearly gain efficiency as the size of the input to the experiments increased, but this trend should be further corroborated. The implications of the speedups are twofold. On one hand, the original codes certainly benefited from being gridified, thus they are appropriate Grid applications to evaluate m-JGRIM. On the other hand, by using policies m-JGRIM achieved competitive speedups levels compared to Ibis and ProActive.

It is worth noting that, although the above experiments treated Ibis and ProActive as competitors of m-JGRIM, these platforms are somewhat complementary to our work. m-JGRIM promotes separation of concerns between application logic and Grid behavior, this latter including Grid services provided by existing platforms. In other words, m-JGRIM provides an alternative method for gridifying applications while it does not "reinvent the wheel" by providing Grid execution capabilities for these applications. In fact, m-JGRIM is currently able to leverage the execution and parallelization services of Ibis, and efforts to integrate it with other projects (specifically ProActive and Condor) are underway. This is, m-JGRIM aims at allowing developers to push the Grid-related code out of the application logic while reusing existing Grid middleware services and still be competitive with these middlewares.

5.2. Analysis of the cost of injecting m-JGRIM metaservices

We quantified the cost of injecting Web Service discovery (Section 5.2.1) and mobility (Section 5.2.2). Experiments were performed on a 1.83 Ghz. PC with 1GB RAM under Linux 2.6.20 and Java 5. The Execution and the Policy subsystems were left out of the analysis as their performance heavily depends on the particular execution services and policies being injected, thus it is difficult to generalize the overhead introduced by them. Besides, as the middleware-level bridge between applications and execution services/policies is just one conventional Spring interceptor, the overhead is intuitively negligible compared to the time required to execute code with these services or process policy code.

5.2.1. Cost of injecting service discovery

We developed two applications for invoking remote Web Services by using Spring remoting and m-JGRIM SDBs. Figure 15 shows the average allocated memory for 20 runs when incrementally invoking these services. For the m-JGRIM implementation, we obtained two variants by enabling and disabling caching, an m-JGRIM feature that allows applications to cache downloaded WSDL definitions and UDDI queries. We took memory snapshots upon initializing the Java and the Spring runtimes (snapshot 1) and after calling 2, 4, 6, 8 and 10 Web Services (snapshots 2 through 6) by using hat [42], a tool to dump the Java heap. Accidentally, the Spring variant initially used more RAM than its m-JGRIM counterparts, because both applications are subject to quite different bean configurations.

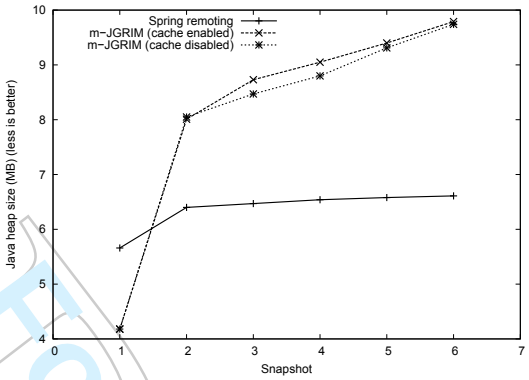


Figure 15: Injecting service discovery: memory usage

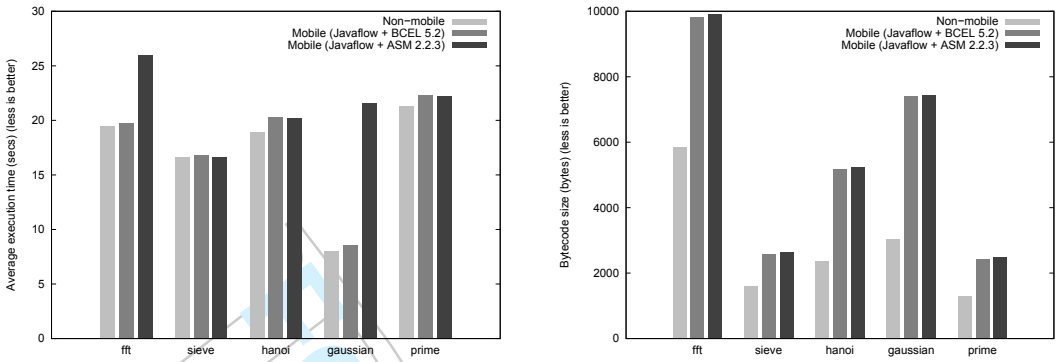
As expected, the memory allocated by the Spring and the m-JGRIM applications behaved linearly. The m-JGRIM variant using caching reduced memory consumption with respect to the variant not using caching by 1%. Moreover, this latter incurred in overheads of 1.5-3.0 MB compared to the Spring solution, which means that maintaining one m-JGRIM SDB requires at the average only 400 KB of extra memory compared to a Spring Web Service proxy (in fact, less when having more SDBs, because they share many factories and internal structures of Spring and m-JGRIM). We believe this small overhead is acceptable as using discovery simplifies configuration (i.e. hardcoding the information to contact services is avoided) and allows for applications that better adapt to the dynamics of Grids (e.g. new services may be available, providers may be temporarily down, etc.).

5.2.2. *Cost of injecting mobility*

The second set of experiments involved the injection of m-JGRIM mobility into five conventional CPU-intensive codes: fast Fourier transform (*fft*), sieve of Eratosthenes (*sieve*), towers of Hanoi (*hanoi*), Gaussian random generator (*gaussian*) and prime number generator (*prime*). All applications were implemented iteratively, totaling 281, 79, 116, 135, and 54 lines of code, respectively. Figure 16 (a) shows the average execution time for 10 runs of the codes on a single PC, and their mobile versions enhanced with Javaflow plus BCEL [43] and ASM [36], two libraries for bytecode instrumentation.

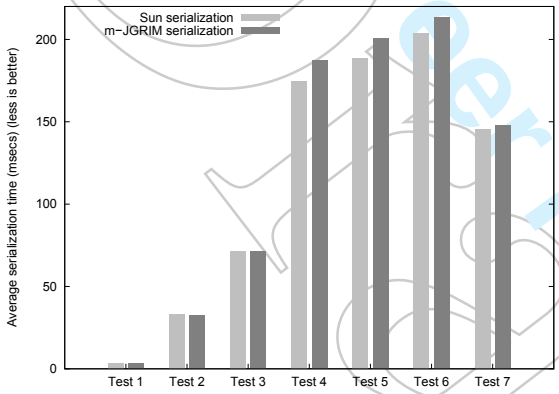
Using BCEL led to a time overhead of at most 7% compared to the non-mobile variants, which is acceptable given the benefits of mobility for exploiting Grids. Nevertheless, this overhead could be cut down by letting developers to selectively instrument only those application methods which use mobility, instead of instrumenting the whole application bytecode. Moreover, we experienced a high performance overhead with ASM when running *fft* and *gaussian*. We found that Javaflow does not fully support ASM, thus further tests should be conducted with an updated version of Javaflow when available. All in all, supporting mobility with BCEL did not incur in excessive execution overhead.

This is the pre-peer reviewed version of the following article: "C. Mateos, A. Zunino and M. Campo: "m-JGRIM: A Novel Middleware for Gridifying Java Applications into Mobile Grid Services". Software: Practice and Experience. Vol. 40, Number 4, pp. 331-362. John Wiley & Sons. 2010. ISSN 0038-0644.", which has been published in final form at http://dx.doi.org/10.1002/spe.961



(a) Average execution time (b) Bytecode size

Figure 16: Injecting mobility: test results



Test	Input data	Size (MB)
Test 1	byte[1000000]	0.95
Test 2	int[1000000]	3.81
Test 3	double[1000000]	7.63
Test 4	Vector<Byte>(100000)	1.48
Test 5	Vector<Integer>(100000)	1.77
Test 6	Vector<Double>(100000)	2.15
Test 7	Vector of 100 WSDL documents	7.91

Figure 17: Performance of m-JGRIM serialization

Figure 16 (b) shows the overhead introduced by mobility in terms of extra bytecode. ASM incurred in overheads of 0.3-2.4% with respect to BCEL. Moreover, the mobile variants added a bytecode overhead of 20-90%, which is a consequence of inserting instructions into the bytecode for strong migration. A negative implication of this is that transferring the instrumented bytecode to remote hosts will require more bandwidth than the non-instrumented one. For larger applications, the required bandwidth could be bigger. Selective instrumentation and/or bytecode compression [44] could alleviate this problem. Another alternative consists of instrumenting application classes at load time. This approach is adopted,

for example, by ProActive. In this way, code transfer is more efficient: the binary code a host receives is just the non-instrumented version, which is then locally altered.

We also compared the performance of m-JGRIM serialization against standard Java serialization. Figure 17 shows the average execution time for 20 runs (all tests used serializable data). The input of Test 7 was 10 copies of the WSDL documents of the services of the previous subsection. Java and m-JGRIM serialization performed similar under tests 1, 2, and 3. Under tests 4, 5, and 6, Java serialization performed about 5-7% better than m-JGRIM, because our mechanism must dynamically check whether each individual object within the input vector is serializable, and in that case, delegate their serialization to Java. Finally, under test 7, m-JGRIM only added an overhead of 2%, even when the amount of in-memory objects of each WSDL was high. The overheads are acceptable for two reasons. First, m-JGRIM offers an alternative to Java serialization for conveniently supporting strong mobility. Second, m-JGRIM is targeted at running resource intensive applications, which by nature spend a large percentage of their lifetime doing useful computations rather than just moving around.

6. CONCLUSIONS

We presented m-JGRIM, a novel middleware for developing Grid applications that materializes JGRIM. The article discussed the design and implementation of m-JGRIM, focusing on how it supports the non-invasive incorporation of Grid functionality such as discovery, mobility and parallelism into ordinary applications. The middleware is based on Java, which makes it portable to various operating systems. In addition, m-JGRIM is fully integrated with standard Web Service technologies.

A major goal of m-JGRIM is to isolate developers as much as possible from the complexities of the Grid while keeping performance and reusability of existing Grid services in mind. We showed its advantages through comparisons with Ibis and ProActive, two Java-based Grid middlewares that materialize alternative approaches for Grid-enabling applications. Roughly, Ibis, ProActive and m-JGRIM were used for gridifying the k-NN algorithm and an application for image restoration. The experiments suggest that m-JGRIM better preserves application logic, which intuitively makes Grid-aware codes easier to maintain, while provides mechanisms to allow gridified applications to perform in a competitive way with respect to related approaches. Despite these encouraging results, we will experiment with more applications and Grid topologies to further validate our middleware.

We also assessed the incidence of the m-JGRIM metaservice layer when executing applications for two common Grid functionalities, namely service discovery and mobility. The results suggest that the overhead in terms of execution time, memory consumption and extra bytecode introduced by m-JGRIM are acceptable, considering the benefits of Grid service injection for simplifying the development of Grid applications [1]. We are enhancing m-JGRIM to further reduce this overhead though.

We are integrating m-JGRIM with other Grid middlewares apart from Satin/Ibis in order to offer a broader variety of services to applications. Besides, as m-JGRIM uses a centralized discovery scheme on top of UDDI that may not be suitable for large Grids, we are extending the P2P facilities of GMAC [41] with decentralized service discovery. In addition, we are redesigning m-JGRIM to support newer Web Service standards (e.g. JAX-WS), and to abstract away its DI container so developers can employ one of their choice. Lastly, we have developed an Eclipse plug-in that lets programmers to gridify their applications by graphically indicating and configuring hot-spots, attaching policies, etc.

REFERENCES

1. Mateos C, Zunino A, Campo M. JGRIM: An Approach for Easy Gridification of Applications. *Future Generation Computer Systems* 2008; **24**(2):99–118.
2. Foster I, Kesselman C, Tuecke S. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *International Journal of High Performance Computing Applications* 2001; **15**(3):200–222.
3. Anderson D, Cobb J, Korpela E, Lebofsky M, Werthimer D. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM* 2002; **45**(11):56–61.
4. Loewe L. Evolution@home: Observations on Participant Choice, Work Unit Variation and Low-Effort Global Computing. *Software: Practice and Experience* 2007; **37**(12):1289–1318.
5. Foster I. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Network and Parallel Computing - IFIP International Conference, Beijing, China*, vol. 3779, Springer, 2005; 2–13.
6. Thain D, Tannenbaum T, Livny M. Condor and the grid. *Grid Computing: Making the Global Infrastructure a Reality*, Berman F, Fox G, Hey A (eds.). John Wiley & Sons Inc.: New York, NY, USA, 2003; 299–335.
7. Natrajan A, Humphrey M, Grimshaw A. The Legion Support for Advanced Parameter-Space Studies on a Grid. *Future Generation Computer Systems* 2002; **18**(8):1033–1052.
8. Grimshaw A, Morgan M, Merrill D, Kishimoto H, Savva A, Snelling D, Smith C, Berry D. An Open Grid Services Architecture Primer. *Computer* 2009; **42**(2):27–34.
9. Atkinson M, DeRoure D, Dunlop A, Fox G, Henderson P, Hey T, Paton N, Newhouse S, Parastatidis S, Trefethen A, *et al.*. Web Service Grids: An Evolutionary Approach. *Concurrency and Computation: Practice and Experience* 2005; **17**(2-4):377–389.
10. Stockinger H. Defining the Grid: A Snapshot on the Current View. *Journal of Supercomputing* 2007; **42**(1):3–17.
11. Mateos C, Zunino A, Campo M. A Survey on Approaches to Gridification. *Software: Practice and Experience* 2008; **38**(5):523–556.
12. Goodale T, Jha S, Kaiser H, Kielmann T, Kleijer P, von Laszewski G, Lee C, Merzky A, Rajic H, Shalf J. SAGA: A Simple API for Grid Applications - High-Level Application Programming on the Grid. *Computational Methods in Science and Technology* 2006; **12**(1):7–20.
13. Bazinet A, Myers D, Fuetsch J, Cummings M. Grid Services Base Library: A High-Level, Procedural Application Programming Interface for Writing Globus-Based Grid Services. *Future Generation Computer Systems* 2007; **23**(3):517–522.
14. Allen G, Davis K, Goodale T, Hutanu A, Kaiser H, Kielmann T, Merzky A, van Nieuwpoort R, Reinefeld A, Schintke F, *et al.*. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE* 2005; **93**(3):534–550.
15. van Nieuwpoort R, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal H. Ibis: A Flexible and Efficient Java Based Grid Programming Environment. *Concurrency and Computation: Practice and Experience* 2005; **17**(7-8):1079–1107.
16. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. *Grid Computing: Software Environments and Tools*, chap. Programming, Composing, Deploying on the Grid. Springer: Berlin, Heidelberg, and New York, 2006; 205–229.
17. Balić B, Bubak M, Wegiel M. LGF: A Flexible Framework for Exposing Legacy Codes as Services. *Future Generation Computer Systems* 2008; **24**(7):711–719.
18. McGough S, Lee W, Das S. A Standards Based Approach to Enabling Legacy Applications on the Grid. *Future Generation Computer Systems* Jul 2008; **24**(7):731–743.
19. Sourceforge. JCGrid. `http://jcgrid.sourceforge.net`.
20. Wrzesinska G, van Nieuwport R, Maassen J, Kielmann T, Bal H. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *International Journal of High Performance Computing Applications* 2006; **20**(1):103–114.
21. GridGain Systems. GridGain. `http://www.gridgain.com`.
22. Alonso J, Hernández V, Moltó G. GMarte: Grid Middleware to Abstract Remote Task Execution. *Concurrency and Computation: Practice and Experience* 2006; **18**(15):2021–2036.
23. Gannon D, Krishnan S, Fang L, Kandaswamy G, Simmhan Y, Slominski A. On Building Parallel and Grid Applications: Component Technology and Distributed Services. *Cluster Computing* 2005; **8**(4):271–277.
24. Fahringer T, Jugravu A. JavaSymphony: A Programming Model for the Grid. *Future Generation Computer Systems* 2005; **21**(1):239–246.
25. Zhang H, Lee J, Guha R. VCluster: A Thread-based Java Middleware for SMP and Heterogeneous Clusters with Thread Migration Support. *Software: Practice and Experience* 2008; **38**(10):1049–1071.
26. van Heiningen W, MacDonald S, Brecht T. Babylon: Middleware for Distributed, Parallel, and Mobile Java Applications. *Concurrency and Computation: Practice and Experience* 2008; **20**(10):1195–1224.
27. Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana S. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing* 2002; **6**(2):86–93.

28. Czajkowski K, Kesselman C, Fitzgerald S, Foster I. Grid Information Services for Distributed Resource Sharing. *10th IEEE International Symposium on High Performance Distributed Computing, San Francisco, CA, USA*, IEEE Computer Society: Washington, DC, USA, 2001; 181–194.
29. Mateos C, Zunino A, Campo M. Extending movilog for supporting web services. *Computer Languages, Systems & Structures* 2007; **33**(1):11–31.
30. Johnson R. J2EE Development Frameworks. *Computer* 2005; **38**(1):107–110.
31. Sun Microsystems. JavaBeans. `http://java.sun.com/products/javabeans`.
32. Rizk P, Kiddle C, Simmonds R, Unger B. Performance of a GridFTP Overlay Network. *Future Generation Computer Systems* 2008; **24**(5):442–451.
33. Milanés A, Rodriguez N, Schulze B. State of the Art in Heterogeneous Strong Migration of Computations. *Concurrency and Computation: Practice and Experience* 2008; **20**(13):1485–1508.
34. Crasso M, Zunino A, Campo M. Easy Web Service discovery: a Query-By-Example Approach. *Science of Computer Programming* 2008; **72**(2):144–164.
35. Apache Software Foundation. Jakarta Commons Javaflow. `http://commons.apache.org/sandbox/javaflow`.
36. ObjectWeb. ASM. `http://asm.objectweb.org`.
37. Lyon D. Simulating Multiple Inheritance in Java. *Concurrency and Computation: Practice and Experience* 2002; **14**(12):987–1008.
38. Mateos C. An Approach to Ease the Gridification of Conventional Applications. PhD Thesis, UNCPBA 2008. `http://www.exa.unicen.edu.ar/~cmateos/files/phdthesis.pdf`.
39. Mateos C, Zunino A, Campo M, Trachsel R. *Parallel Programming and Applications in Grid, P2P and Networked-based Systems*, chap. BYG: An Approach to Just-in-Time Gridification of Conventional Java Applications. Advances in Parallel Computing, IOS Press: Amsterdam, The Netherlands, 2009. To appear.
40. Sun Microsystems. Java Management Extensions (JMX). `http://java.sun.com/products/JavaManagement`.
41. Gotthelf P, Zunino A, Mateos C, Campo M. GMAC: An Overlay Multicast Network for Mobile Agent Platforms. *Journal of Parallel and Distributed Computing* 2008; **68**(8):1081–1096.
42. Sun Microsystems. Heap Analysis Tool (hat). `https://hat.dev.java.net`.
43. Apache Software Foundation. Byte Code Engineering Library (BCEL). `http://jakarta.apache.org/bcel`.
44. Stefanov E, Sloane A. On the Implementation of Bytecode Compression for Interpreted Languages. *Software: Practice and Experience* 2009; **39**(2):111–135.