JADiMA: Virtual Machine Architecture for building JAVA Applications on Grid Platforms.

Yudith C. Cardinale and Eduardo A. Blanco and Jesús De Oliveira
Universidad Simón Bolívar, Dept de Computación y T.I
Caracas 1080-A, Venezuela, Apartado 89000
{yudith,eduardo}@ldc.usb.ve, jesus@bsc.co.ve

Abstract

This paper describes JADiMA (*Java Distributed Machine*), a collaborative platform to construct high performance distributed JAVA applications. JADiMA is a system that automatically manages the remote libraries used in a JAVA application. JADiMA takes the advantages of portability, modularity, flexibility and object oriented model of JAVA, while incorporating well known techniques of communication and security. The result is a simple and efficient distributed environment upon which applications and data are easily shared and highly portable amongst heterogeneous platforms and multiple users. JADiMA allows compilation and execution of JAVA applications which use distributed libraries, without the need of keeping them either in the developer or the user hosts. To illustrate the functionality and characteristics of JADiMA, we show examples of constructing real applications with several levels of library package dependencies in distributed environments.

Keywords: Management of distributed libraries, Reusability, Compilation with distributed libraries, Distributed execution, Grids platforms, High performance JAVA applications.

1 INTRODUCTION

Recent developments in the use of distributed resources for the execution of high performance applications has increased the possibility of collaborative environments in which multiple users, geographically distant, can share data, software, computing resources, and even specialized devices [BFH03, Abb04]. Regarding this, it is common today to use several software libraries developed by third parties to attain together the global goal required by an application. Following the reusability principle, it is more practical for developers to leave the application

specific functionalities on pieces of software intensively proved and developed exclusively for those specific functions, and so, they can focus on solving their main objective.

In this way, developers must search and find adequate libraries and reference them in their code so that the compilation process can be successful. Besides, end users must have the same libraries (the same versions used in the compilation) so that the application can execute adequately. In general, applications are packed together with the libraries on which they depend. This implies that the reused pieces of software must be kept locally to the compilation and execution environment. In distributed environments this approach presents some strong disadvantages: i) waste of disk space, when the same library is used for several applications or when only a small portion of the code of such library is used; ii) difficulty in handling and updating library versions, due to that responsibility of updating local libraries with more recent improved versions is left in hands of developers and end users; and iii) in the specific case of developing applications for computational grids, ideally, libraries are only required for local compilation as the execution is always carried out in one or some remote nodes of the computational grid. The fact of downloading the libraries locally just for compilation is a waste of space and time for the developer.

Programming languages such as C++, Fortran and HPF have been traditionally preferred by developers of scientific applications. However, JAVA has recently introduced itself as an attractive language for application programming in general because of its simple and clear object model. Especially for collaborative, heterogeneous, distributed environments, a portable execution platform, such as the JAVA Virtual Machine, makes the implementation of scientific applications potentially easier. Multi thread support and concurrence, combined with the inherent portability of JAVA is especially interesting for parallel and distributed scientific applications. Even though JAVA is not a very appropriate language nowadays to carry out high performance computation, the international scientific community is making important efforts to achieve a performance similar to that reached with more traditional languages as Fortran or C++. In fact, there is already evidence that shows JAVA has the potential to achieve the performance of such traditional scientific languages [BSPF01, Mor98]. The use of JAVA as high performance scientific application programming language is been promoted through the JavaGrande Forum [Jav98]. This Forum is focusing the efforts to improve those JAVA aspects dealing with high performance computation including compilers and libraries.

In this paper, we introduce JADIMA (*Java Distributed Machine*), a collaborative platform for building high performance distributed JAVA applications. JADIMA is a system that automatically manages remote libraries used by a JAVA application. JADIMA exploits the advantages of portability, modularity, object oriented

model, and flexibility of JAVA, while uses well known communication and security techniques (such as the SOAP protocol and the use of X.509 certificates). The result is an efficient and simple distributed computational environment on which applications and data are easily shared and highly portable among heterogeneous platforms and multiple users, which results in the reuse and sharing of libraries among novel and advanced programmers.

JADIMA allows the compilation and execution of high performance JAVA applications that use distributed software components without keeping them locally in the application developer machines. In this sense, JADIMA lets developers use, during the compilation process, a much smaller representation of the libraries (referred to as *stubs*) on which the application depends. *Stubs* are generated automatically by JADIMA and substitute real libraries. During the execution time, actual class definitions are downloaded from very well known library repositories as long as they are been required by the application. The use of a version numbering convention allows the automatic updating of new versions of used libraries without affecting the application correct execution. To illustrate the functionality and characteristics of JADIMA, we show examples of constructing real applications with several levels of library package dependencies in distributed environments.

The rest of this paper is organized as follows. Section 2 presents the general design of JADIMA architecture. In section 3, we describe the JADIMA's communication and security schemes. A comparative study with related work is presented in 4. The description of how JADIMA works is presented in Section 5 through examples of constructions of real applications. Finally, in Section 6, we summarize our conclusions and propose further possible extensions to our research.

2 DESCRIPTION OF JADIMA ARCHITECTURE

JADIMA's design is carried out to satisfy the following requirements:

- Easy to install, configure and use. We want JADIMA to work without following difficult steps or extensive configuration processes. Users should not require technical knowledge for its installation and use. All users should be able to benefit from the advantages of JADIMA.
- Flexibility, adaptability and modularity. By designing the framework into layers and defining general interfaces in all JADIMA components, JADIMA allows integration of different well known, standard communication, security and data access mechanisms.
- Platform Independence. JADIMA is fully implemented in JAVA. Since the JAVA language and libraries are platform-independent, JADIMA can be in-

stalled, with no need of recompilation, in any platform that has a JAVA run-time environment.

- **Transparency.** Neither the applications nor the libraries will have to be modified to benefit from the functionalities of JADIMA.
- **High performance and scalability.** In order to enhance the applications performance, JADIMA provides mechanisms for class caching and pre-fetching, thus reducing the impact of remote class transfer during execution.
- **Security.** The code and data exchange between clients and libraries repositories must be secure. JADIMA guarantees that all downloaded code comes from reliable repositories, and that every client that downloads a library is authorized. In addition, library access control was implemented to satisfy the privacy amongst different user groups. To enforce security during execution, the applications run in a *sandbox*, preventing them from stealing or damaging local data.
- **Accessibility.** Users have access to a large range of libraries without worrying about their physical location.

In the following sections, we describe in detail how JADIMA architecture satisfies these requirements.

2.1 JADIMA Components

The application development process proposed by JADIMA is defined according to the identification of the following tasks: library administration, publication, compilation and execution of applications that require these libraries. Each of these tasks is represented by a component: the *Repository*, the *Compilation Agent*, and the *Execution Agent* respectively. Figure 1 presents a general outline of JADIMA architecture.

2.1.1 Repository

This component is in charge of the administration of remote libraries that will be used in compilation and execution processes defined in JADIMA. For each published library, the *Repository* will keep three sets of data:

1. implementation of actual classes given by the publishing user which will be used in the execution phase.

Figure 1: JADiMA Architecture

2. reduced versions of libraries (*stubs*), which will only be used during the application compilation process.
3. library documentation

The Repository includes three elements that allow separating the physical medium of the library storage and recovery operations. The first element is called the Data Access Abstraction Layer, which only defines the operations allowed for administering the libraries (as publishing, consulting and querying information) and does not depend on the physical medium for storage. The second element, the Data Access Module implements the defined operations in a direct way with the third element, which is a physical storage medium, through a specific manager such as MySQL, DB2, Oracle or a traditional file system. This last component is an external entity to the definition of JADIMA (See figure 2).

2.1.2 Publication Agent

Publishing a library in JADIMA consists of two phases:

Figure 2: Logical view of JADiMA Repository

1. Generation of a reduced representation of the library (*stubs*). A *stub* is generated for each library class: first, a class skeleton is created; this is the definition of all method signatures in a . java source file; second, these definitions are compiled using the JADiMA Compilation Agent, in order to resolve the dependencies that these libraries may have with other previously published libraries. Figure 3 shows the *stub* generation process. *Stubs* will be used by developers in the application compilation phase.
2. Data transmission (*stubs*, documentation and library packages) from the publication node to the Repository.

The instruction that causes all this process has the following form:

Figure 3: Generation of *stubs*

```
jdm_publish -b <library_name>.jar -v <major>.<minor>.<revision>
            -r <repository_name> -n <library_name> -j <javadoc_zip_file>
```

where:

-b <library_name>.jar, refers to the library package that user wish to publish.

-v <major>.<minor>.<revision>, is the version number assigned. Publication of a library requires the developer to assign a version number based on a three-digit scheme ¹ in which:

- **major**: represents changes in external functionalities and in method signatures.
- **minor**: refers to minor functionality changes, there are no changes in method signatures.
- **revision**: there are no changes in functionality or method signatures, corresponds to corrections on previous library distributions.

¹Our proposed version number scheme can be adapted according to the package version numbering system proposed by Sun Microsystems [SM02].

-r <repository_name>, is the name of the Repository. Repository contact information is found in a xml configuration file (for example, names and respective urls). This configuration file may be generated with the help of a graphical interface.

-n <library_name>, it is the name that will identify the library in the Repository.

-j <javadoc_zip_file>, indicates the .zip file name which contains the library documentation.

2.1.3 Compilation Agent

The Compilation Agent is in charge of obtaining the information given by the developer about the application dependencies and requests the library *stubs* to the specified Repositories. With the help of a graphical interface, the programmer may consult information about the libraries published on different Repositories, and select those of his/her interests. After the developer selects the libraries needed for the applications, the graphical interface automatically generates the metadata file, which describes the dependencies of his/her project. During the compilation process, it is not required to download the published classes. The *stubs* allow for the compilation process to be successful are obtained, even without the presence of actual libraries. The compilation instruction has the following form:

```
jdmc -s <project's sources package root> -d <class files destination
      directory> <javac param1> .... <javac paramN>
```

where:

-s <project's sources package root>, specifies the directory where the application class source files are located.

-d <class files destination directory>, specifies the directory in which the *stubs* will be located and the classes compiled.

<javac param1> <javac paramN>, can be traditional javac options.

2.1.4 Execution Agent

The Execution Agent is responsible for establishing an environment in which the application can be executed in a transparent way. When the execution is requested, the user has a compiled application using JADIMA (which contains a

stub for each library used) and a dependency file, which is used to initiate the `jdmClassLoader`. When the application code references any of the classes belonging to one of the remote libraries, the `jdmClassLoader` finds and downloads, from the `Repository`, the actual definition of the class substituting the *stub*. It is necessary to highlight that this scheme allows to download, into the execution platform, only the library classes actually used by the application, and also allows to obtain improved versions of the libraries that will not affect the execution (according to our proposed version numbering scheme). The execution instruction is:

```
jdm -p <project's package root dir or jar> -m <main class name>
      <param1> ... <paramN>
```

where:

-p <project's package root dir or jar>, is the directory or .jar file of the application to be executed.

-m <main class name>, specifies the name of the main class.

<param1> ... <paramN>, are the application parameters or those from the Virtual Machine.

Figure 4 shows a graphical outline of the execution process.

Class Prefetching and Caching

In order to reduce the communication time, and thus to improve the performance during the execution, we have implemented a scheme of class pre-fetching. The `Execution Agent` keeps information representing associations amongst the application classes according to a temporal relationship. This relationship defines sets of classes referenced within a time interval. Whenever an application is executed, this information is updated by averaging the elapse times at which a class is referenced, from the beginning of the execution. With this information, the `Execution Agent` defines *clusters* of classes referenced within a time interval of δ . Hence, when a class is referenced and is not present in the execution node, all the classes that belong to its *cluster* are requested. Information of class associations is returned to the client at the end of the execution, such that it can be submitted in future executions. Class pre-fetching module is defined as a thread, so it can be executed concurrently with the application. In this way, we enable overlapping of class transfers with computation.

Figure 4: JADiMA Execution Model

The Execution Agent manages the persistence of the downloaded classes in a local cache. This means that after execution, remotely loaded classes remain in the Execution Agent. In this way, it is possible to reduce communication delay and overhead in subsequent executions. The implementation of cache policies are considered with a *plug-in* scheme that reinforces JADiMA adaptability. We defined an interface with generic access methods to the cache, that can be used to implement different class replacement policies. The default policy offered at the moment is “Last Recently Used” (LRU).

3 COMMUNICATION AND SECURITY SCHEMES IN JADiMA

Communications among JADiMA components is based on *plug-ins* that allow the implementation and use of any communication protocol or mechanism. JADiMA components interact with the Communication Abstraction Layer (see figure 2), which is a well defined interface that abstract the communication model out of its underlying implementation. Different modules attached to this interface are in charge of carrying out effective communication among Agents and Repositories for the data exchange. In this sense, implementing a new communication scheme consists on developing a module which implements the Communication Abstraction Layer using the desired communication mechanism.

The exchange of information among `Agents` and `Repository` must be carried out on safe environments which provide user authentication mechanisms and protection of communication channel. JADIMA users must register with a determined role describing the actions that are permitted on the system. Besides, users must also have digital credential for authentication effects. The currently defined roles are: i) publication, to carry out publication, querying, and access to libraries; ii) administration, to carry out administrative activities on `Repositories`; iii) version and library access, to restrict access to library groups. With each request `Repository` receives the user role (previously authenticated according to the authentication mechanism implemented at the level of the `Communication Module`) with which the `Repository` carries out authorization verification.

The current communication mechanism used in JADIMA is based on SOAP over HTTPS using Apache Axis, but it can be substituted by others such as CORBA, RMI, HTTP or sockets due to the layered design of JADIMA. User authentication is delegated on the Tomcat application server (servlet container) that uses mutual authentication mechanism with X.509 digital certificates. Assigning an authenticated user to a security role is accomplished according to the configured mechanism on the Tomcat application server which keeps information of the users credentials and their roles. This security information can be manage on relational databases, plain files or LDAP directories.

4 RELATED WORK

Specialized library reusability for the development of high performance scientific applications is an advantage that can be attained in collaborative environments such as computational grids. Yet, there are still limitations on the tools that support the compilation, deployment, distribution and execution processes of those applications in grid systems. JADIMA outperforms many of these limitations and provides a simple, easy-to-use Virtual Machine architecture for the construction of JAVA applications in distributed platforms.

Apache Maven [Apa05] and Krysalis Centipede [Kry04], are projects that extend traditional compilation tool functionality (such as `ant` and `make`) for the distributed management of libraries. Just like JADIMA, these environments consider that libraries remain in web server repositories. However, from the construction file (or metadata file) where the programmer describes the dependencies, libraries are downloaded in local directories which replicate the repository structures. JADIMA offers graphical interfaces that facilitates the metadata file generation in xml format, from which only the corresponding library *stubs* will downloaded locally. Furthermore, as JADIMA can integrate several communication protocols, reposi-

tories do not necessarily have to be web servers. Using Apache Maven and Krysalis Centipede, version updates can be carried out during the compilation time only if it is explicitly specified in the dependency file. In the case of JADIMA, however, version updating is carried out during execution time. Based on the proposed version numbering system, the `jdmClassLoader` is able to download last versions or revisions of the referenced classes without affecting the execution. JADIMA can manage fine-grained security by allowing access control at the library level and not at the repository level. In Apache Maven and Krysalis Centipede, security is managed in the communication layer. None of these two tools give support to execution.

DistAnt [GA04] and GridAnt [AvLaMHZ⁺04] are systems that extend the ant file construction environment to provide an automatic procedure of deployment application for its execution on distributed platform. The deployment process includes file transfer, installation on particular computing resource, and configuration for its later execution. DistAnt and GridAnt can be combined with Apache Maven to obtain the benefits in the compiling process, still keeping previously mentioned disadvantages that Apache Maven has. Similar to JADIMA, GridAnt uses X.509 certificates to offer authorization and authentication (with single sign-on). The application construction files in those projects are based on xml format and must be produced manually by developers or users specifying the grid computational resources on which the applications will be executed. In JADIMA, the metadata file is generated automatically and the process of contacting the respective repositories is transparent to the programmer. For the execution, it will be the grid platform which selects the appropriate computing resources and the `jdmClassLoader` is in charge of locating and downloading the real classes in a transparent way. Both DistAnt and GridAnt are administrator oriented tools to install software in a remote manner, while JADIMA is oriented to support scientific application development activities.

Several projects which extend the JAVA Virtual Machine to facilitate parallel and distributed application execution in grid environments have been proposed. Some examples of those projects are: Addistant [TSCI01], Unicorn [OLLY02], Javelin++ [NBK⁺99], JNLP [Zuk02], Bayanihan [SH99], HORB [Sat97], and SUMA/G [CH05]. None of those projects provide support to the compilation process. Even though most of this research projects have different purposes to those of JADIMA, they are of relevance as complementary tools in the realm of execution.

Addistant allows to adapt an application developed to be executed in a unique Virtual Machine, so it can be executed in a distributed environment (this is functional distribution). This execution model is exactly contrary to the interests of JADIMA, which consists on executing, in one computational resource, an application whose parts could be distributed. In this sense, JADIMA does not modify

the original application execution model. In Addistant, developers specify where instances of each class will be executed in a “policy” file. From this file, it generates proxy classes, equivalent to *stubs* in JADIMA, which will be in charge of carrying out remote references. The application is automatically modified in load time to substitute the references of the classes that are not local, by references to the respective proxy classes. JADIMA performs the substitution at execution time.

JNLP (JAVA Network Launching Protocol), is a project that is quite similar to JADIMA with respect to the execution model. JNLP uses mechanisms related with a Classloader to load classes that compose an application from a web server during the execution time. However, it is not transparent in many aspects: i) applications must be programmed with particular specifications of JNLP specific packages; ii) the developer must specify a metadata file with the application properties and the modules required by the application; iii) in case any of the modules changes, this file must be updated manually; and iv) JNLP requires installing a plug-in in the browser to recognize the metadata file and begin the application module partial download. This is the only project considering class prefetching, but it is the developer who specifies which classes can be loaded together, while JADIMA makes class prefetching in a transparent way.

JADIMA execution model allows it to be easily integrated to those execution platforms, and in this way attain a much more complete support environment to compilation and execution.

5 CASE STUDIES: TIE and JUNG

To illustrate concepts that JADIMA handles and demonstrate its functionality, we carried out tests with real applications of open source code with different dependency levels among the used packages. In this section we will refer to the experience with TIE [Sie03] (*Trainable Information Extractor*) and JUNG [OFNK03] (*Java Universal Network/Graph*). TIE is a trainable software for information extraction and text classification. JUNG is an open-source software library that provides a language for the modeling, analysis, and visualization of graph and network data. Dependencies of the libraries used by these two applications are shown in table 1.

The tests were carried out in an scenario where JADIMA components were distributed in nodes in different domains with different environments and platforms. Laboratory resources of Universidad Simón Bolívar (USB) and external resources from this university were used. These resources had the following characteristics:

- Pentium III, 800 MHz with 256 MB of RAM, 10 GB in Disk, in Computation Laboratory (LDC) of USB.

Application Name	Used Libraries	Description
TIE	commons-beanutils commons-collections commons-configuration commons-discovery commons-lang commons-logging commons-math commons-pool dom4j jtidy junit minorThird velocity	http://jakarta.apache.org/commons/beanutils/ http://jakarta.apache.org/commons/collections/ http://jakarta.apache.org/commons/configuration/ http://jakarta.apache.org/commons/discovery/ http://jakarta.apache.org/commons/lang/ http://jakarta.apache.org/commons/logging/ http://jakarta.apache.org/commons/math/ http://jakarta.apache.org/commons/pool/ http://www.dom4j.org/ http://jtidy.sourceforge.net/ http://www.junit.org/ http://minorthird.sourceforge.net/ http://jakarta.apache.org/velocity/
minorThird	BeanShell MontyLingua CRF COLT Gauss-Newton and Conjugate-Gradient optimization JCommon JFreeChart JUnit JWF LBFS Libsvm Log4J Trove	http://www.beanshell.org/ http://web.media.mit.edu/~hugo/montylingua/ http://crf.sourceforge.net http://cern.ch/hoscchek/colt/ http://billharlan.com/pub/code/inv/indexh.html http://www.jfree.org/jcommon http://www.jfree.org/jfreechart/index.html http://www.junit.org http://sourceforge.net/projects/jwf/ http://crf.sourceforge.net/ http://www.csie.ntu.edu.tw/~cjlin/libsvm/ http://jakarta.apache.org/log4j/docs/ http://trove4j.sourceforge.net/javadocs/
JUNG	COLT commons-configuration	http://cern.ch/hoscchek/colt/ http://jakarta.apache.org/commons/configuration/

Table 1: Library Dependencies

- Dual Pentium III, 800 MHz with 512 MB of RAM, 10 GB in Disk, in the Parallel and Distributed Systems Laboratory (SPD) in the USB
- AMD Athlon XP, 3.1 GHz, with 1 GB of RAM, 80 GB in Disk, and dual Dell of 800 MHz, 512 MB of RAM, 7 GB in Disk, in external nodes to the USB (BSC Consultores C.A).

All nodes are connected to Internet through links of 256 Kbps in average. The tests carried out in this scenario were the following:

- An LDC Repository, with Apache Tomcat 5.0.16 on Fedora Core 3, with a RDBMS Data Access Module using the MySQL database located in another node of the same domain.
- One Repository in a node of BSC Consultores C.A. (BSC Repository), executing JBOSS Application Server 4.0.0 on RedHat Linux 9, with a RDBMS

Data Access Module using a Microsoft SQLServer 2000 database located in another node of the same domain. The operating system of these nodes is Microsoft Windows 2000 Server.

- One Repository in SPD Laboratory (SPD Repository), with Apache Tomcat 5.5.9 on Scientific Linux CERN 3, with a Data Access Module through NFS.
- Publication, compilation and execution nodes in SPD Laboratory, with Scientific Linux CERN 3.
- Publication, compilation and execution nodes in LDC with Fedora Core 3
- One execution node in a portable device connected to a domainless net with Microsoft Windows 2000 Professional.

In all nodes, we used JAVA Virtual Machine 1.5.0. The test strategy is as follows:

1. Publication of libraries used by the applications (these are those specified in column 2 of Table 1, except for `minorThird`). Different versions of these libraries were published randomly among the three Repositories, from different publication nodes. The publication process implies the generation of the respective *stubs* and documentation, along with transferring the data (*stub* documentation and the libraries themselves) from the publication node towards the specified Repositories. All this process can be realized with a simple instruction as the following:

```
jdm_publish -b ./commons-discovery.jar -v 1.6.0 -n commons-discovery
             -r BSC_REPOSITORY -j commons-discovery-javadocs.zip}
```

It is necessary that the configuration file is located in the publication node. This configuration file has the contact information to the Repositories (for instance, names and their respective urls), and can be automatically generated with the help of a graphical interface.

2. Compilation of the `minorThird` library in a SPD compilation node. Using a graphical interface, the different libraries are selected, published in the different Repositories from which `minorThird` depends and the dependencies file is generated. To compile the following command is used:

```
{\tt jdmc -s /home/yudith/minorThird/src -d /home/yudith/minorThird/build
          -Xlint:none}
```

In this phase, the respective *stubs* are downloaded so that compilation can be successful. It is necessary that the configuration file with contact information to the *Repositories* is located in the compilation nodes. In this way, the *Compilation Agent* connects to the three *Repositories* to solve dependencies.

3. Publication of the *minorThird* compiled version in LDC *Repository*. The process is similar to that explained in step 1.
4. Compilation of the “TIE” application in a compilation node of LDC. The process is similar to that explained in step 2.
5. TIE “demo” execution in an external execution node with the command:

```
jdm -p /home/yudith/tie/build -m de.fu.berlin.ties.Main class=train
      -outdir=. -classifier=Winnow ../*.dsv
```

The *Execution Agent* first downloads the *stubs* that are not present in the local cache (*stubs* may not be present in the execution node because compilation was executed in another node or because the *stubs* were replaced in the cache), later, the *jdmClassLoader* is instantiated to load the main class. As long as the other classes are being referenced, the *jdmClassLoader* is in charge of downloading the real classes taking into consideration the version number that is specified in the respective *stub*.

6. JUNG application compilation in a LDC compilation node. In this case the *Compilation Agent* is only connected with the *Repositories* where the COLT libraries and *commons-configuration* were published.
7. JUNG “demo” execution from the same node in which it was compiled.

The successful realization of these tests shows the functionality of JADIMA and illustrates how easy it is to be used.

6 CONCLUSIONS Y FUTURE WORK

With JADIMA we extended JAVA Virtual Machine potentiality to a computational grid environment. As a consequence, developers and users are provided with a collaborative simple and efficient platform which allows to easily share software resources and motivates code reusability. JADIMA is in charge of the automated management of remote libraries used by a JAVA application. The compilation and execution mechanisms implemented allow that applications and data could be easily shared and highly portable among heterogeneous platforms and multiple users.

The modular and layered design of JADIMA provides for easy incorporation of well known communication, data access, and security mechanisms. The results of our exploration show a clear positive vision of the feasibility and practicality of distributed compilation and execution environments with automated management of remote libraries. Currently, we are working on the integration of JADIMA on SUMA/G, a JAVA based computational grid. With this experience, we hope to provide a mechanism that permits easy integration of JADIMA environment to any grid platform that provides remote executions for JAVA applications.

References

- [Abb04] Ahmar Abbas. *Grid Computing: A Practical Guide to Technology and Applications*. Charles River Media, 2004.
- [Apa05] Apache Software Foundation. Apache Maven Project, 2005. <http://maven.apache.org/>.
- [AvLaMHZ⁺04] Kaizar Amin, Gregor von Laszewski and Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Albert Rossi. GridAnt-A Client-Controllable Workflow System. In *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.
- [BFH03] F. Berman, G. Fox, and A. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003.
- [BSPF01] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, Palo Alto, California, United States, 2001. ACM Press.
- [CH05] Y. Cardinale and E. Hern´andez. Parallel Checkpointing on a Grid-enabled Java Platform. *Lecture Notes in Computer Science*, 3470(EGC2005):741 – 750, February 2005.
- [GA04] Wojtek Goscinski and David Abramson. Distributed Ant: A system to support application deployment in the grid. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID’04)*, 2004.
- [Jav98] Java Grande Forum. Java Grande Forum Report: Making Java work for high-end computing. Technical Report

- JGF-TR-1, Java Grande Forum Panel, 1998. Available at <http://www.javagrande.org/sc98/sc98grande.pdf>.
- [Kry04] Krysalis Community Project. Krysalis centipede, 2004. <http://krysalis.org/centipede/>.
- [Mor98] Jos´e Moreira. Closing the Performance Gap between Java and Fortran in Technical Computing. In *Java for High Performance Computing Workshop, Europar 98*, 1998.
- [NBK⁺99] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: Scalability issues in global computing. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 171–180, San Francisco, California, June 1999.
- [OFNK03] Joshua O’Madadhain, Danyel Fisher, Tom Nelson, and Jens Krefeldt. JUNG: Java Universal Network/Graph Framework, 2003. <http://jung.sourceforge.net/index.html>.
- [OLLY02] T. M. Ong, T. M. Lim, B. S. Lee, and C. K. Yeo. Unicorn: voluntary computing over Internet. *ACM SIGOPS Operating Systems Review*, 36(2):36–51, April 2002.
- [Sat97] Hirano Satoshi. HORB: Distributed Execution of Java Programs. In *Proceedings of the International Conference on Worldwide Computing and Its Applications*, pages 29–42, March 1997.
- [SH99] Luis F.G. Sarmenta and S. Hirano. Bayanihan: building and studying web-based volunteer computing systems using JAVA. *Future Generation Computer Systems*, 15(5–6):675–686, October 1999.
- [Sie03] Christian Siefkes. TIE: Trainable Information Extractor, 2003. <http://www.inf.fu-berlin.de/inst/ag-db/software/tie/index.html>.
- [SM02] Inc. Sun Microsystems. Package version identification, 2002. <http://java.sun.com/j2se/1.5.0/docs/guide/versioning/index.html>.
- [TSCI01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A Bytecode Translator for Distributed Execution of ”Legacy” Java Software. In *Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001)*, volume 2072, Budapest, Hungary, June 2001. Springer-Verlag.

- [Zuk02] John Zukowski. Deploying Software with JNLP and Java Web Start, August 2002.
<http://java.sun.com/developer/technicalArticles/Programming/jnlp/>.