Middleware Support for the Deployment of Resource-Aware Parallel Java Components on Heterogeneous Distributed Platforms

Yves Mahéo, Frédéric Guidec, Luc Courtrai

Middleware Support for the Deployment of Resource-Aware Parallel Java Components on Heterogeneous Distributed Platforms

Yves Mahéo, Frédéric Guidec, Luc Courtrai
VALORIA Laboratory, Université de Bretagne-Sud, France
{Yves.Maheo|Luc.Courtrai|Frederic.Guidec}@univ-ubs.fr

Abstract

This paper reports the development of the Concerto platform, which is dedicated to supporting the deployment of resource-aware parallel Java components on heterogeneous distributed platforms, such as pools of workstations in labs or offices. Our work aims at proposing a basic model of a parallel Java component, together with mechanisms and tools for managing the deployment of such a component on a distributed platform. Moreover, we strive to provide components with means to perceive their runtime environment, so they can for example dynamically adapt themselves to changes occurring in this environment. The Concerto platform was designed in order to allow the deployment of parallel components on a distributed platform. It additionally defines and implements an open and extensible framework for distributed resource discovery and monitoring in such an execution environment.

1. Introduction

There is a growing demand for distributed applications that can be ported on a wide variety of platforms, including the distributed platforms obtained by assembling regular workstations in labs and company offices. However, developing distributed applications from scratch is known to be a difficult and tedious task, requiring much knowledge and expertise. The component-oriented approach can help significantly with this respect. This approach promotes the development of so-called "software components", that can later serve as deployment units, and that can further be assembled while developing more complex components, or full-featured applications. Although the advantages of this approach are now widely admitted, little effort has been invested so far on the development of components specifically designed for distributed platforms.

In our work we experiment with the development of "parallel components", that is, components that are meant to be deployed on a distributed platform, and that each encapsulate a parallel code to be run on this kind of platform. Moreover, since these components we consider are liable to be deployed and executed on possibly unstable and heterogeneous platforms, we strive to provide these components with means to perceive their execution environment, so they can adapt dynamically to changes observed in this environment. By doing so, we try to foster the development of *adaptive* parallel components, that is, components whose behavior is not defined statically, but components that can instead adjust their behavior dynamically in order to react to environmental variations.

Although component adaptability is our prime objective, this paper does not focus on this topic per se. Instead it presents a Java-based middleware platform we have designed in order to provide support for such components. The platform Concerto implements a number of facilities for deploying, launching, and controlling the execution of parallel components on a distributed platform. To achieve these goals it uses facilities we have implemented in D-RAJE *(Distributed Resource-Aware Java Environment),* an open and extensible framework that makes it possible to monitor the state of the many resources offered in a distributed platform. Figure 1 shows the overall organization of the Concerto platform.

The remaining of this paper is organized as follows. Section 2 gives an overview of D-RAJE. Section 3 presents the basic model of a parallel component we propose and implement in Concerto, and it shows how this model can be used to implement a demonstrator parallel component that encapsulates an adaptive, distributed genetic algorithm. This section also describes the facilities we implemented for managing the deployment and the execution of parallel components on a distributed platform. Section 4 discusses related projects on distributed software components. Section 5 concludes the paper.

2. Support for resource modeling and monitoring

One of the main objectives of project Concerto is to provide parallel software components with means to perceive their runtime environment, so that they can adapt to variations observed in its characteristics. Environment modeling and monitoring in distributed systems has already been addressed in numerous projects pertaining for example to Grid computing or to network computing (such as [5, 11, 10, 14]). In many of these projects, though, resource modeling is performed at a rather coarse grain (*e.g.,* computing node, global free memory, network link), and information about resources is often limited to system resources, and collected using *ad hoc* methods. An important part of Concerto is made of a software framework called D-RAJE. This platform was designed so as to offer an extensible model in which any kind of resource in a distributed environment can be reified as a Java object. It also provides various facilities for monitoring these resources in a homogeneous way. A brief description of D-RAJE is given below. A more detailed presentation can be found in [9].

2.1. Resource modeling

With D-RAJE a distributed system can be modeled and monitored using Java objects that reify the various resources offered in this system. As a general rule, we qualify as "resource" any hardware or software entity a software component is liable to use during its execution. To date the types of resources considered in D-RAJE –and thus in the Concerto platform– include system resources (CPU, system memory, swap, network interfaces, etc.) that chiefly characterize the underlying hardware platform, as well as so-called "conceptual" resources (sockets, processes, threads, directories, files, RMI servers, etc.) that rather pertain to the applicative environment in which a component is running. An extensible hierarchy of classes models these resources. Any resource object (that is, any instance of one of these classes) is capable of producing on demand an observation report, which provides a summary of its state.

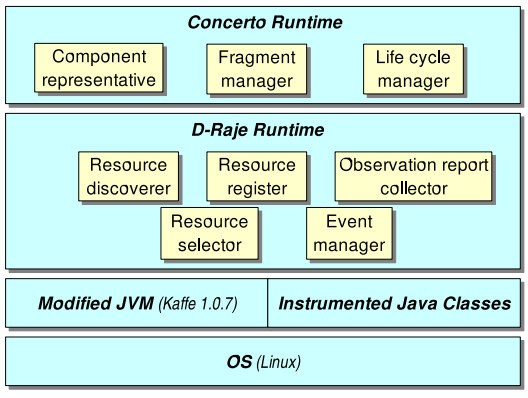

Figure 1. Overview of the Concerto platform (on one host)

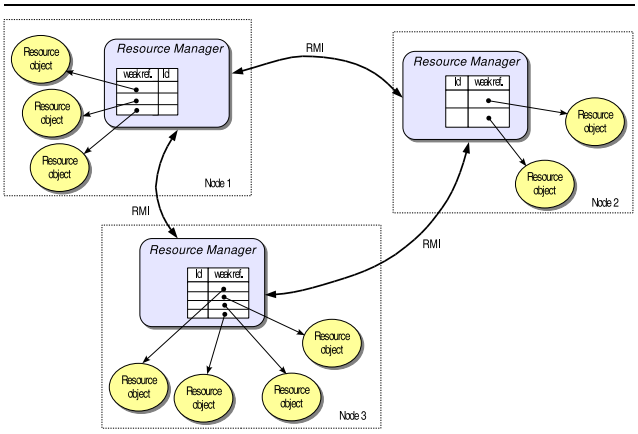

Figure 2. Architecture of the distributed resource manager, which keeps track of all resource objects created on the various hosts (or nodes) of the distributed platform.

From the viewpoint of an application programmer, resource objects can be created explicitly, or one can rely on a resource discovery monitor that is able to automatically instantiate (and destroy) resource objects pertaining to some classes of resources.

At creation time, references to resource objects are added to a local resource register. A resource manager is distributed on the distributed platform so as to play the role of a global register. This resource manager makes it possible to access information about distributed resources in a homogeneous way, regardless of where these resources are actually located in the system. Figure 2 shows the general architecture of the distributed resource manager. Local instances of this manager are created on each host, and these instances interact with one another thanks to the Java RMI mechanism. Moreover, each instance maintains a weak reference to any resource object created locally, so it can observe the state of any local resource at any time, and provide remote instances of the manager with similar information whenever required. Weak references are used –instead of standard references– so as not to prevent application components from deleting resource objects when these objects are not needed anymore at application level.

D-RAJE somehow compares with the Common Information Model [8] as far as resource modeling is concerned.

However, D-RAJE goes beyond pure modeling, as it additionally implements facilities for collecting and exploiting information about the state of any system or conceptual resource (please see [9] for details on this topic).

2.2. Resource selection and observation

The resource manager can be consulted when looking for a specific kind of resource in the distributed system, or in order to request information about the status of any resource in this system. Since the population of resource objects can be quite large, a selection can be performed based on their location (*e.g.* on a specific host, in the whole distributed system, or in neighboring hosts), their type, or any other criterion. Resource filtering is obtained by using so-called resource patterns. Any programmer can define new patterns, although many patterns are already available as predefined patterns in D-RAJE. With these patterns, one can for example identify all objects modeling host CPUs, or network interfaces, or TCP sockets. These objects can additionally be searched for on a specific host, or in the whole distributed platform. One can even perform a more selective search, and look for example for TCP or UDP sockets that have been bound to a local port whose number is in a specific range (say, values between 0 than 1024), and that maintain connections with remote hosts whose IP addresses conform to a specific pattern (say, all addresses in domain 195.83.160/22). Resource identification can thus be achieved at quite a fine grain if needed.

Once a resource has been identified, information on its status can be obtained through any one of two observation mechanisms. The first of these mechanisms allows direct observation of the resource considered: the resource manager can be asked to collect and return an observation report concerning a specific resource, wherever this resource is located in the system. The second observation scheme relies on an event-based model: local or distant monitors are automatically created when needed. These monitors can observe the resource considered periodically, and notify the user when user-specified events occur. A thorough description of these topics can be found in [9].

3. Parallel Components

The Concerto platform is dedicated to supporting adaptive parallel components. Our aim is to propose a basic component model suited for parallel and distributed applications, and to provide an infrastructure that enforces the adaptability of components. We propose below a definition of what we mean by parallel component in Concerto.

A programmer who wishes to develop a parallel component for Concerto must design his component as a set of cooperating threads. He must also define the business part of the component (name, interface, implementation). The platform offers facilities for managing non-functional aspects of the component.

3.1. Component model

3.1.1. Component interface Any parallel component hosted by the Concerto platform must define the following interfaces.

- *Business interface*: No constraint is put on the type of the business interface. The programmer of a component may for example propose an interface based on Java RMI. The component is then an object implementing the *Remote* interface, whose methods are to be called remotely by the component's clients (Section 3.2 describes a demonstrator component that has been designed along this line). The component may also behave as a server listening to a specific TCP or UDP port, with which a client program can communicate. Alternatively, one may design a distributed interface (that is to say, an interface associated with several objects that each implement a part of the interface), so the component can be connected in parallel with another parallel component.
- *Life-cycle interface*: Through this interface, the different steps of the life of the component can be controlled. To date, this covers deploying the component on the distributed platform, launching all the activities (*i.e.,* starting the threads) of the components, and stopping these activities. In the future we may include persistence services in this interface.
- *Resource interface*: The component exhibits a resource interface through which the user can access information related to the resources used by the component. Actually, the component itself is considered as a resource. Hence any component in Concerto must implement a method *observe()* that returns an observation report. By default, the observation report generated by a component simply aggregates the observation reports of all the resources it itself uses. For security reasons, the component's programmer may find it desirable to change this default implementation, for example by restricting the amount of information disclosed to the component's client.

3.1.2. Internal structure of a component When building a parallel component, the programmer develops a set of Java threads (actually, a set of classes implementing the *Runnable* interface). These threads cooperate to perform functions specified in the component's business interface.

Threads are grouped in placement (or distribution) entities called fragments. A fragment is a set of threads that belong to a single component, and that must run within

the same virtual machine on the same distributed platform host. The threads of a fragment can thus share a common object space. Communication and synchronization between such threads can be performed just like in any other multi-threaded Java program. On the other hand, threads that belong to different fragments must rely on external communication and synchronization mechanisms, such as sockets and RMI.

Information on the component's structure is available through facilities implemented in D-RAJE, since threads, fragments and the component itself are modeled as resources in D-RAJE. As already mentioned, some of this information is also available to other components through each component's resource interface.

3.1.3. Component deployment The deployment of an application on a distributed platform is known to be a tedious and error-prone task. In order to alleviate the deployment of parallel components on such platforms, we have developed an XML dialect that permits a component programmer to describe the component's structure, placement directives for its fragments, and constraints imposed for the deployment to be feasible (availability of a specific version of the JVM, of a RMI registry, etc.)

The Concerto platform includes an administration tool with which a distributed platform can be managed, using either a command line or a graphical user interface. Figure 3 shows an example of what can be displayed by the graphical interface. With this administration tool, hosts can be added to or removed from the distributed platform at any time. A parallel component can be loaded in the platform by providing a *.jar* file that aggregates Java code for the component's threads, an XML deployment descriptor, and possibly additional documentation and data associated with the component. During this loading operation, a new *Component* resource object is created in D-RAJE, and automatically registered with the resource manager. Once a component has been loaded, it can be considered as a new resource in the distributed platform. Another component can therefore trigger its deployment and activation on the platform. Alternatively, these operations can be triggered by any administrator –or user– of the platform thanks to the above-mentioned administration tool.

3.2. Example

This section illustrates the utilization of the Concerto platform for building a genetic algorithm (GA) component that can be deployed on a heterogeneous distributed platform. In this particular example the GA component aims at finding an arithmetic expression that approximates a function that can be numerically evaluated. Of course our GA component deals with a population of individuals that are submitted to genetic operators such as mutation and crossover. A selection mechanism makes it possible to identify the best individuals in the population, so that they can participate in the production of new individuals.

3.2.1. Business interface In order to develop a Concerto component based on this genetic algorithm, the programmer must first specify the business part of the component. In this particular case, the business interface is the interface of a RMI server. As illustrated in Figure 4, this interface offers methods for initializing the population, for launching the reproduction process, for obtaining the current generation, and for halting the reproduction process.

```
import java.rmi.*;
public interface GeneticAlgorithm extends Remote {
  public void setInitialPopulation(Population pop)
          throws RemoteException;
  public void startReproduction()
          throws RemoteException;
  public Population getCurrentGeneration()
          throws RemoteException;
  public void stopReproduction()
          throws RemoteException;
}
```

Figure 4. Business interface of a Concerto component.

3.2.2. Parallel implementation The master-slave paradigm is used for implementing a parallel version of the algorithm. Notice that in this particular case our aim is not to produce the best possible parallel genetic algorithm, but simply to illustrate the utilization of Concerto. Therefore, no in-depth optimization has been applied to the original parallel algorithm, nor to its parallel implementation. The population in the GA component is to be distributed on a set of distributed hosts, which will be considered as slave hosts. Each slave will deal with its own sub-population, and will make it evolve through crossovers, mutations, and selections. A particular host, called the master, will periodically gather all the sub-populations, perform a selection phase on the whole population, and scatter it again over the slave hosts.

The resulting GA component is thus composed of a set of slave threads, conducted by a unique master thread. Each of these threads is placed in a distinct fragment. Communication between the slaves and the master is performed through RMI calls. In order to simplify the architecture, the master thread also behaves as an RMI server, implementing the business interface of the GA component.

3.2.3. Deployment Deployment directives are specified in a Concerto component descriptor that comes in the form of an XML file (see Figure 5). When the deployment of the

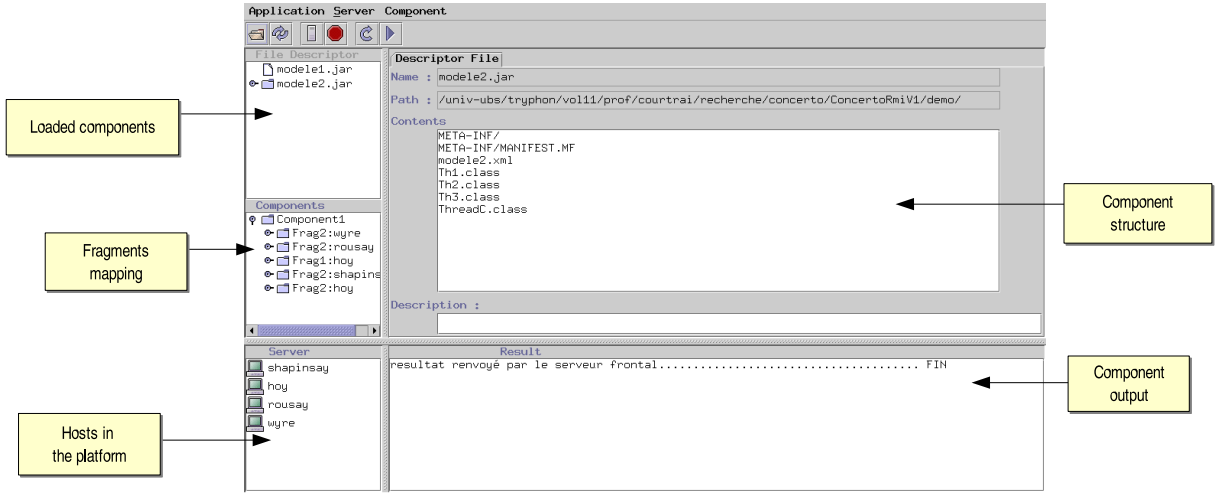

Figure 3. Snapshot of the Concerto graphical administration tool

genetic algorithm component is triggered –for example via the Concerto administration tool–, a succession of operations is performed that consists in analyzing the XML descriptor, checking the requirements specified (in this case, verifying the presence of an RMI registry), loading classes *ThMaster* and *ThSlave* on the different JVMs, and creating all the fragments and threads (that will consequently be registered as resources in D-RAJE). These threads will actually be started when the component is activated.

3.2.4. Adaptation A first, straightforward adaptation of the algorithm can be performed by having the master take the actual number of slave threads into account. This number is determined during the deployment phase, based on the actual number of hosts in the distributed platform at deployment time. In the future we plan to insert a new placement directive (ALL-ALWAYS) in our XML dialect, so that replicated fragments can be automatically created and started when hosts are inserted in the distributed platform while a parallel component is running.

Figure 6 shows how the resource manager can be asked to identify the slave fragments. In this example a resource pattern called *FragmentPattern* is used in order to look for resources that are actually component fragments. Moreover a parameter is passed to this pattern in order to select only those fragments named “GA-Slave”.

The real adaptation of the algorithm is performed by allowing for the heterogeneity of the hosts on which the slave threads are running, as well as for the variability of the CPU workload on these hosts. Indeed, we cannot assume that the CPU type is identical on all hosts in the distributed platform, and that these hosts are all dedicated to running only the GA component. The time a slave thread can spend applying genetic operators on its sub-population may thus vary a lot. As a consequence, the master should be able to tune the size of the sub-population it sends to a particular slave based on the computation power that can be expected from this slave. The CPU power available on a given host can be deduced from the CPU frequency of this host, and from the amount of time the CPU remains idle during an observation period (a period of one second was chosen for this particular ap-

```
<?xml version=”1.0” encoding=”ISO-8859-1” ?>
<component>
  <description>Genetic Algorithm Component</description>
  <name>
    <concerto-name>GenAlgoRes</concerto-name>
    <rmi-name>GeneticAlgorithmServer</rmi-name>
  </name>
  <requirements>
    <java kaffe=”1.0.7” d-raje=”1.0”/>
    <concerto-vers=”1.0”/>
    <rmiregistry/>
  </requirements>
  <file>GenAlgorithm.jar</file>
  <fragment-list>
    <fragment name=”GA-Master”>
      <node>ANY</node>
      <thread-list>
        <runnable-class=”ThMaster”/>
      </thread-list>
    </fragment>
    <fragment name=”GA-Slave”>
      <node>ALL</node>
      <thread-list>
        <runnable-class=”ThSlave”/>
      </thread-list>
    </fragment>
  </fragment-list>
</component>
```

Figure 5. Example of an XML deployment descriptor.

```
ResourceManager rm
    = ResourceManager.getResourceManager();

Set slaveIdSet
    = rm.getResourceIds(new FragmentPattern("GA-Slave"),
                        new GlobalSearch());
```

Figure 6. Illustration of how the resource manager can be used to look for specific resources in the whole distributed platform. In this example specific component fragments are sought.

plication). Information about each CPU (including its frequency) can be obtained by collecting reports from CPU objects in D-RAJE. Likewise, D-RAJE makes it possible to implement and to deploy monitors that observe the state of a given resource periodically. With such monitors, calculating how much computing power is used on a host of the distributed platform is straightforward.

3.2.5. Measurements We have conducted a set of preliminary measurements in order to evaluate the impact of the utilization of Concerto –and that of the adaptation facilities it offers– on the overall performance of our GA parallel component. In this experiment the population consisted of 1024 individuals. At each iteration step, each slave computed 10 generations before sending back its sub-population to the master. The overall process iterated 100 times. Table 1 shows the execution times obtained for a varying number of hosts. During this experiment we used evenly loaded 2.4 GHz Pentium 4 PC hosts running Linux 2.4.20. The speedup remains acceptable for the distributed platform sizes considered. The impact of adaptation is showed in Table 2. In this case a slower host (400 MHz Pentium PC) was included voluntarily in the distributed platform, and a random load was put on every CPU by running dummy processes. In the case of one host, the table shows the execution times observed depending on whether the slow host was used, or one of the fast hosts. Executions involving more than one host systematically implied the slow host, plus one or several additional fast hosts. Experiments were performed with and without component adaptation. Results show that a significant gain is obtained when adaptation is performed.

4. Related Work

The originality of the Concerto approach lies in the fact that it combines a parallel component model together with some support for distributed resource observation. Component-based application development is already possible through the use of "standard" component technologies, such as Microsoft COM [12] or Sun's Enterprise Java Beans. The OMG (Object Management Group) also developed its own solution with the CORBA Component Model [13]. However these technologies were not specifically designed to support parallel components, that is, components that inherently involve parallel activities. Few works have been carried out on models or platforms that target parallel and distributed components. One of them is the Common Component Architecture [1], which defines a component model suited for parallel scientific applications. In this architecture, emphasis is put on the definition of a scientific interface definition language (SIDL), and the specification of ports that permit component interoperabilty. The main objective in CCA is to allow the efficient interoperability of pre-existing scientific codes. Like in Concerto, few constraints are put on the way the functional part of a parallel component is implemented. However, no support is provided in CCA for component adaptation. Project Padico [7] bears similarities with CCA as it is dedicated to coupling scientific codes. It extends the Corba Component Model, and it relies on a specific communication toolkit in order to provide efficient communication between several SPMD parallel codes encapsulated in components. Some form of adaptation is provided in the communication layer, which allows for several commonly used communication libraries.

# of slaves	Execution time
1	582 sec.
2	381 sec.
3	297 sec.
4	221 sec.

Table 1. Execution times of the GA on a homogeneous distributed platform

The work described in [2] also focuses on parallel and distributed components: an implementation of the Fractal [3] hierarchical component model is proposed in order to provide so-called Grid Components. This implementation is based on the Pro-Active library [4]. It provides facilities for asynchronous communication, migration of activities, deployment and debugging. Like in Concerto, a parallel component may be implemented according to the MIMD model. Introspection on both the components structure and placement is possible, though both kinds of introspection rely on distinct mechanisms.

None of the above-mentioned works proposes a homogeneous way to get precise information about the environment of a component (including the system), and about its sub-components.

In Concerto, we strove to associate Java facilities for

# of slaves	Execution time with adaptation	Execution time without adaptation
1 (fast host)	400 sec.	407 sec.
1 (slow host)	2112 sec.	2150 sec.
2	344 sec.	1344 sec.
3	234 sec.	974 sec.
4	203 sec.	806 sec.

Table 2. Impact of adaptation on a heterogeneous distributed platform

resource observation with the implementation of an open component model suited for adaptive application development. In this respect, we share a similar goal with the framework for adaptive components described in [6]. This framework is based on Fractal and it defines context-awareness services. Well-designed adaptation mechanisms are built in this framework, but no specific support for distribution aspects is provided.

5. Conclusion

In this paper we have presented the Concerto platform, which aims at allowing the deployment and the support of parallel components on distributed platforms. Ongoing work focuses on proposing a basic parallel component model, as well as tools for the deployment of such components. In a first stage, our objective is to adopt as simple and flexible a model as possible.

The distributed platforms targeted by the Concerto platform are primarily those composed by enlisting non-dedicated, heterogeneous, and possibly unstable workstations in labs or offices. The runtime environment offered to parallel components is therefore liable to exhibit much heterogeneity and dynamicity. For this reason, the Concerto platform implements facilities for discovering how resources are distributed in the distributed platform, and for monitoring these resources at runtime.

The development of the Concerto platform is still in progress. Moreover a number of parallel components are also being developed. In the near future we plan to use the platform for deploying instances of these components on different kinds of distributed platform. A series of experiments will then be run, in order to evaluate the benefits of designing parallel components that can monitor their environment, and that can adjust their behavior to variations observed in this environment.

Acknowledgments

This work is supported by the French Ministry of Research in the framework of the ACI GRID programme.

References

[1] R. Amstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-Performance Scientific Computing. In *8th Int. Symposium on High-Performance Computing*, Redondo Beach, California, Aug. 1999.

[2] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *Proc. of International Symposium on Distributed Objects and Applications (DOA'2003)*, LNCS, Catania, Sicily, Nov. 2003.

[3] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proc. of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, June 2002.

[4] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13), Nov. 1998.

[5] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *10th IEEE Int. Symposium on High-Performance Distributed Computing*. IEEE Press, August 2001.

[6] P.-C. David and T. Ledoux. Towards a framework for Self-Adaptive Component-Based Applications. In *4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'2003)*, Paris, France, Nov. 2003.

[7] A. Denis, C. Pérez, T. Priol, and A. Ribes. Padico: A Component-Based Software Infrastructure for Grid Computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS'2003)*, Nice, France, Apr. 2003. IEEE Computer Society.

[8] DMTF. CIM specification v2.2. Technical Report DSP0004, Data Management Task Force, http://www.dmtf.org 1999.

[9] F. Guidec, Y. Mahéo, and L. Courtrai. A Java Middleware Platform for Resource-Aware Distributed Applications. In *2nd Int. Symposium on Parallel and Distributed Computing*, Ljubljana, Slovenia, Oct. 2003.

[10] F. Kon, R. Campbell, M. D. Mickunas, K. Nahrstedt, and F. J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *9th IEEE Int. Symposium on High Performance Distributed Computing*, Pittsburgh, USA, Aug. 2000.

[11] K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Dis-

tributed Computing. *Sofware – Practice and Experience*, 32(2):135–164, Feb. 2002.

[12] Microsoft. The Component Object Model Specification. Technical report, Microsoft Corporation, Oct. 1995.

[13] OMG. CORBA Components. Technical Report OMG-orbos-99-07-01, OMG TC Documents, July 1999.

[14] F. Sacerdoti, M. Katz, M. Massie, and D. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proc. of Int. Conference on Cluster Computing*, Hong Kong, Dec. 2003.