A Java Middleware Platform for Resource-Aware Distributed Applications

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

Abstract

*This paper reports the development of D-*RAJE (Distributed Resource-Aware Java Environment)*, a Java-based middleware platform that makes it possible to model and to monitor resources in a distributed environment. With this middleware, any kind of hardware or software resource can be modelled using standard Java objects, and services allow to discover local as well as remote resources, and to observe the state of these resources either locally or remotely. D-*RAJE *is meant to ease the development of adaptive, security-oriented, or QoS-oriented Java applications, as well as the development of platforms capable of supporting such demanding applications.*

***Keywords** : Java, resource awareness, support for adaptation, distributed environments and applications.*

1 Introduction

Nowadays distributed applications must face an ever-growing diversity of runtime conditions. For many such applications, the hardware devices on which application components are liable to be deployed cover a wide spectrum, ranging from embedded or portable devices with limited capabilities, up to full-featured, powerful workstations.

It is our conviction that the development of platforms capable of supporting adaptive programs should significantly contribute to alleviate the burden of application programmers, when they strive to design portable, yet efficient distributed applications for heterogeneous distributed systems. The services offered by operating systems or at middleware-level on such a variety of devices are unfortunately quite disparate. In such conditions, ensuring the portability of a distributed application and supporting the interoperability of its many components become crucial objectives. Any application component should be able to allow for the nature and characteristics of the resources available on the platform it has been deployed on.

Java has proved to be an interesting approach for hiding heterogeneity in distributed systems. The JVM provides a standard runtime environment for Java applications. The characteristics of the underlying hardware platform and the operating system are either masked or perceived through standard programming interfaces. However, although it is clearly an advantage for developing portable code, the level of abstraction offered becomes a drawback when it comes to developing adaptive application, since such applications should be allowed to take benefit from information pertaining to the resources provided by the execution environment: the execution environment is hidden so well that collecting information on this environment becomes quite difficult. Indeed, operating systems usually provide tools for collecting information about the resources they manage. Unfortunately, all types of resources are not managed the same way. The picture is even less favorable when considering a distributed system, where all nodes do not necessarily run the same operating system. There is thus a need for mechanisms that permit an homogeneous access, at the Java level, to resource information across a distributed platform.

The perception of the execution environment forms the basis of adaptation decisions. This perception is usually achieved at a coarse grain: for example, load balancing relies on the CPU load on each node, or persistence functions are conditioned by free disk space. We believe that fine grain perception could also be useful: a multi-threaded application could for instance take benefit from information on the memory consumption of a specific thread instead of allowing only for the system free memory amount. Similarly, knowing the number of bytes sent on each individual socket could give a better insight on network utilization than the overall amount of data sent through a network interface. We believe that both kind of information should be obtained through a unique framework.

This paper presents an overview of D-RAJE *(Distributed Resource-Aware Java Environment)*, an open, object-oriented middleware platform with which a dis-

tributed system can be modelled and monitored using Java objects that reify the various resources offered in this system. Our main objective is to develop some middleware that makes it possible for higher-level pieces of software (eg application programs) to perceive their runtime environment, so that they can adapt their behaviour to the characteristics of this environment, and to possible dynamic variations in these characteristics. As a general rule, we qualify as "resource" any hardware or software entity a software component may use during its execution. The resources considered to date in the platform include system resources (CPU, system memory, swap, network interfaces, etc.) that chiefly characterize the underlying hardware platform, as well as other resources (sockets, processes, threads, directories, files, RMI servers, etc.) that rather pertain to the applicative environment considered. Besides allowing that resources be modelled and handled as Java object, D-RAJE implemements mechanisms that make it possible for applications to discover the existence of a specific resource (or kind of resource) in their environment; to search for a specific resource (or kind of resource) in their environment; to ask for the condition (ie state) of a specific resource or to be notified when changes occur in the state of a resource.

Since D-RAJE is dedicated to resource modelling and monitoring in a distributed system, the dissemination of resources in this kind of environment must be allowed for. The above-mentioned mechanisms are thus implemented in such a way that all resources can be identified and monitored in a homogeneous way, regardless of where they are located.

The remaining of this paper is organized as follows. Related work is discussed in Section 2. Section 3 explains how resources are modelled in D-RAJE. Mechanisms offered for the distributed management of resources are described in Section 4. Section 5 details some implementation choices. Lastly, Section 6 summarizes the paper and mentions cases of D-RAJE utilization.

2 Related work

Resource modelling and monitoring has justified much effort in the past few years. Considering only fine-grain resources on a per host basis, some of the services D-RAJE offers for modelling and monitoring resources compare with those offered by JRes [5], GVM [2], and KaffeOS [1]. These works are mostly devoted to providing a secure environment for application programs based on variations –or extensions– of the traditional sandbox security model (as implemented for example in the standard Java Runtime Environment). However monitoring is limited to a pre-defined set of general resource categories (such as the network and the filesystem). Our approach differs in that D-RAJE primarily defines a framework for resource modelling and monitoring, and provides a number of generic tools and facilities in order to ease the integration and the support of resources of any kind.

As far as the distribution of resources is concerned, several works have been –or are being– carried out in the context of network computing or grid computing. D-RAJE generally does not share the same overall objective and differs in the granularity of the resources handled. In the industrial area, the Data Management Task Force proposes a Common Information Model [7], a set of object oriented models that form the basis of several products for managing systems and networks across multiple organizations. In the context of grid computing, many projects propose tools for modelling and monitoring the resources scattered in a large scale distributed environment [9]. They often rely on distributed directory services (eg. Globus [6], Condor [12]) or follow an object-oriented approach (eg. Legion [3], Javalin [11]) to store and retrieve resource information. The resources considered in these environments are mainly coarse-grain resources, such as computing nodes or storage units. The objective is to be able to collect information about disseminated resources in order to manage the allocation of some of these resources for the execution of a given set of computational-intensive jobs. The distributed operating system 2K [8] aims at providing an adaptable architecture for developing and deploying distributed services across heterogeneous platforms. Resource management in 2K relies on Corba and resource allocation based on the needs of application components is considered as a part of the resource management process. The scope of D-RAJE is somehow more restricted, as its purpose is to serve only for distributed resource modelling and monitoring. Services specifically dedicated to resource allocation, resource reservation or QoS provision can be built on top of D-RAJE, but their definition and their implementation remain outside the scope of this project [10].

Our work with D-RAJE is thus focused on providing an easy-to-use, extensible Java framework for resource modelling and monitoring in a distributed environment. Its main purpose is to support distributed Java applications that can adapt themselves to –possibly fine-grain– resource information.

3 Resource modelling

The architecture of D-RAJE is organised around a class hierarchy in which a Java class is defined for each resource type. Figure 1 shows some of the classes we have defined so far. The classes shown in this figure either reify typical system or hardware resources (CPU, memory, etc.), as well as other resources that are not necessarily identifiable as hardware or system parts (such as an RMI registry, or an RMI server).

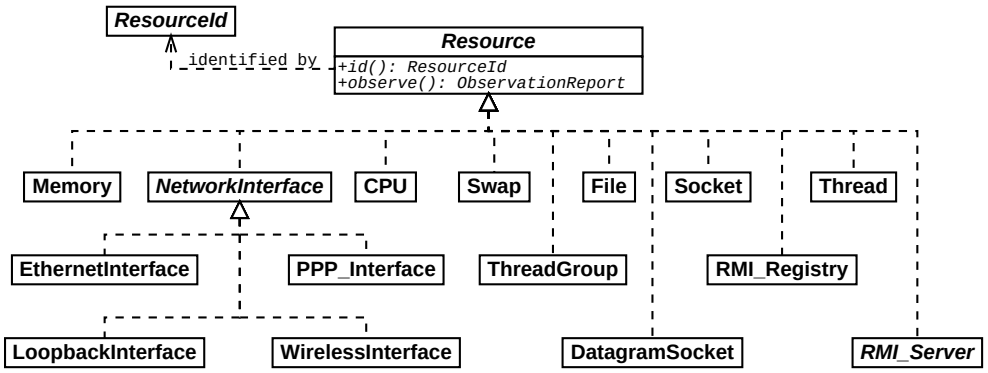

Figure 1. Object-based modelling of common system and non-system resources.

By modelling all kinds of resources in D-RAJE, our prime objective is to provide Java programs with means to perceive their runtime environment. We also permit that programs identify, locate, and monitor these resources using the generic facilities implemented in D-RAJE.

The class hierarchy used for modelling resources in D-RAJE was designed so as to be easily extensible. As a consequence, any new kind of resource can be included in the hierarchy at any time and with minimal effort. D-RAJE should actually be perceived as a design framework, rather than on a pre-defined toolkit or Java package. Similarly, the classes we have developed so far should simply be considered as examples, as they are meant to demonstrate how resources can be reified and then handled as Java objects in D-RAJE.

As shown if Figure 1, a class meant to reify a resource type in D-RAJE must implement the *Resource* interface. This interface serves as the root of the hierarchy of resource classes, but it also specifies that any resource object should have a unique identifier, and that it should be capable of producing an observation report on demand. Resource identification and observation reporting are important topics in D-RAJE. They are discussed in details in sections 3.2 and 4.1.

3.1 Reification of resources

Detailed information about low-level resources is usually not available to programmers of Java applications. One of our objectives in this project is to bring forth such information through the boundaries of the JVM, so that Java programmers can take benefit from this kind of information.

Each of the classes reifying system resources in D-RAJE defines methods for consulting the state of the actual resource it models. For example, by calling appropriate methods on an instance of class *Memory* one can consult the amounts of free and used physical memory in the system. The abstract class *NetworkInterface* similarly defines a set of methods for consulting the state of any kind of network interface, and descendants of this class provide additional methods for consulting attributes that only make sense for specific kinds of interfaces (such as the number of collisions registered by an Ethernet interface, or the radio channel currently used by a wireless interface).

The notion of ”resource” can actually be extended far beyond that of basic hardware or system parts. Many software elements can also be considered as resources. Indeed, as soon as a piece of software provides a specific service that can be used by other elements (such as entire application programs, or other pieces of software), then it too can be considered as a ”resource”. For example, an RMI server is a resource for potential RMI clients. Similarly, when considering a deployment platform where transmissions through a network can somehow be restricted (based for example on the range of accessible remote IP addresses and ports), each transmission socket can in turn be perceived as a resource. We sometimes use the terms ”conceptual resource” or ”applicative resource” to denote those resources that cannot be directly or systematically identified as hardware or system parts, but that definitely provide a service that may be of interest to application programs.

3.2 Observation reports

Any object that implements the *Resource* interface defines specific methods that allow the consultation of the current state of the resource it models. However each resource type defines a very specific set of methods. This approach makes it difficult to monitor heterogeneous sets of resources using a single, generic scheme. Observation reports were defined in D-RAJE so as to circumvent this constraint. An observation report is meant to capture and to preserve information about the state reached by a given resource at a given time. A hierarchy of Java classes was developed (see Figure 2) so as to define different kinds of observation reports,

and facilities implemented in D-RAJE allow the generation, the collection, and the management of such reports.

A specific kind of report must be defined for each resource type. As a consequence the hierarchy of observation report classes somehow mirrors that of resource classes, and most of the methods defined in resource classes in order to consult the state of the resources they model have a counterpart in the corresponding observation report classes. The method *observe()* declared in interface *Resource* (see Figure 1) provides a generic access point for requesting observation reports from resources of any kind. Any resource class defined in D-RAJE must implement this method. This approach makes it possible for application programs to request reports about resources using a generic scheme (by calling method *observe()*), regardless of the actual type of the resources considered, and regardless of the type of the reports thus obtained.

3.3 Sporadic system resources

Any kind of resource can be modelled as a Java object in D-RAJE. To achieve this goal the first condition is that a Java class be defined for this kind of resource. The second condition is that one or several instances of this class be created at runtime in order to model the resources that actually exist in the system.

Fulfilling the second condition may in some cases be quite difficult for system resources. Consider an application programmer who wishes to monitor the state of each network interface in the system. If this system is a workstation with a permanent connection to a LAN infrastructure, then there is probably no ambiguity as to which network interface must be modelled as a Java object. The programmer simply needs to know the name of the device associated with this interface at system level (such as "eth0", see line 1 in the example below), and create a resource object accordingly (all *NetworkInterface* classes take the name of a network device as a construction parameter). Now, if the system is actually a mobile laptop, then this system may show a highly dynamic behavior as far as network connectivity is concerned. In such a case where system resources can appear and disappear dynamically, there is a need for tools that can help the programmer discover system resources and keep informed about the creation and deletion of sporadic system resources. A few such tools have already been implemented in D-RAJE for specific resource types. For example a monitor was implemented, whose role is to poll the system periodically in order to keep informed about network interfaces. This monitor maintains a population of *NetworkInterface* objects, and whenever a network interface appears or disappears in the system the monitor updates this population accordingly. Line 2 and 3 in the example below illustrates the use of this kind of monitor.

```
// (1)   Create a known resource
Resource r = new EthernetInterface("eth0");
// (2)   Create a monitor
ResourceDiscovery rd = new EthernetDiscovery();
// (3)   Get the set of current network interfaces
Set s = rd.getResources();
```

As a general rule, similar monitors can be designed to monitor any kind of resource whose persistence in the system is not guaranteed (eg PCMCIA or USB devices). In the future we plan to design specific tools for interacting with plug-and-play mechanisms, such as the Linux HotPlug package.

4 Dealing with distributed resources

D-RAJE handles every *Resource* object created on any host of the distributed platform. This handling is performed by the main component of D-RAJE called the "resource manager". This component is spread over the distributed system and is accessible to the user through an instance of the *ResourceManager* class. The resource manager provides several services across the distributed platform: it registers every creation or destruction of a resource object, identifies and locates every resource object and offers some facilities for obtaining information about resources.

The current implementation of the resource manager is fully distributed. Each of the *ResourceManager* objects that contribute to the resource manager function maintains a registry of the resources created locally and cooperates with its counterparts using RMI.

4.1 Resource identification and registration

Resource objects can be created "manually" in an application program, or a pool of resource objects can be handled automatically or semi-automatically thanks to resource discovery facilities such as those described in Section 3.3. The mechanisms we designed can help identifying and tracking resources at runtime. They rely on a naming system that gives any resource a unique identity. Whenever a resource object is created, the constructor requests a unique identifier (object of type *ResourceId*, see Figure 1) from a *ResourceIdGenerator*. It then registers with the resource manager.

D-RAJE mostly proposes a framework for dealing with resource objects at runtime. Various implementations can fit in this framework, so that quite complex resource identification and registration systems (such as hierarchical ones) could very well be deployed. As a first step, the current implementation provides a flat naming scheme.

When a resource becomes of no interest for an application, it is likely that the application eventually maintains no reference to the resource object. As it desirable that this kind of object be garbage collected, the resource manager

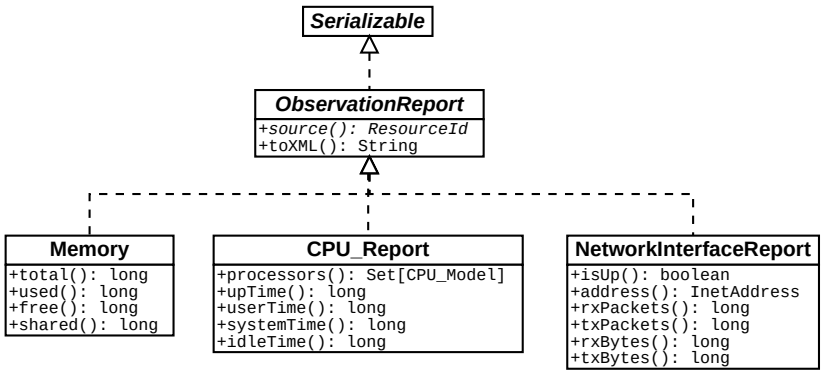

Figure 2. Observation reports modelling.

only uses weak references to resources objects so garbage collecting is possible.

4.2 Resource tracking

The number of resources in a distributed system may be quite high. We defined a collection of patterns that make it possible to look for resources, or to collect observation reports from resources, selectively. The interface *SearchPattern* serves as the root of a hierarchy of classes that each describes a specific search strategy (Figure 3). Search patterns are meant to help focus on the resources located on a fraction of a distributed system.

The following segment of code shows how a resource manager can be requested to look for resources using different search patterns. In this example several patterns are used in order to specify that the search should apply (line 1) to local resources only; (line 2) to the resources located on a remote node whose identity is specified as an argument; (line 3) to all resources, wherever they are.

```
ResourceManager manager = ResourceManager.getManager();
// (1)
Set localIds = manager.getResourceIds(new LocalSearch());
// (2)
Set remoteIds =
      manager.getResourceIds(new LocalSearch(remoteNodeId));
// (3)
Set allIds = manager.getResourceIds(new GlobalSearch());
[...]
// (4)
ObservationReport report = manager.getObservationReport(resId);
```

In this example, the method *getResourceIds()* is invoked on the resource manager. This method returns a set of identifiers of all known resources. One can then require that the resource manager collect and return an observation report concerning one of these resources (line 4, assuming that the value of *resId* was extracted from one of the three id sets).

4.3 Resource classification and selection.

The resources registered within a resource manager can be of various types (eg *CPU, Memory, Swap,* etc.). D-RAJE implements mechanisms for classifying and selecting resources based on the notion of "resource pattern".

The interface *ResourcePattern* (see Figure 3) defines a function *isMatchedBy()*, which takes a resource object as a parameter, and returns a boolean whose value depends on whether this object satisfies the considered selection criterion or not. In the most simple scenario resource selection can simply be based on the actual type of the resource object which is submitted to the test. More sophisticated selection mechanisms can also be implemented using the resource pattern mechanism. For example the class *NetworkInterfacePattern* permits the selection of network interfaces based on various criteria. Of course this pattern will only select *NetworkInterface* objects, but it can also be used more selectively so as to match, for example, only those network interfaces that are multicast capable, or those that currently run in promiscuous mode.

The following example shows the creation of two resource patterns. The first pattern permits the selection of any *EthernetInterface* object. In this example the constructor of the pattern is called with no parameter, which means that any *EthernetInterface* object should be selected by this pattern. The second pattern makes it possible to select only those resource objects that model sockets resources, and that additionally satisfy the following selection criteria: the IP address of the remote host must belong to the 195.83.160/24 network, and the remote port must be in the range 0 to 1023. On the other hand, the local IP address and port the socket is bound to can take any value.

```
ResourcePattern ethPattern = new EthernetInterfacePattern();
ResourcePattern socketPattern =
      new SocketPattern(InetAddress.AnyAddress, "195.83.160/24",
            PortRange.AnyPort, new PortRange(0, 1023));
```

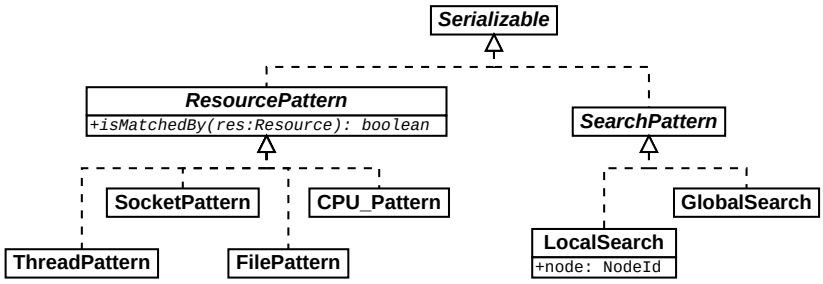

Figure 3. Modelling of the patterns that can be used to select resources (left-hand side of the hierarchy) and to describe search strategies (right-hand side of the hierarchy).

The resource manager can handle requests that take a *ResourcePattern* or a *SearchPattern* object (or both) as a parameter. One can thus request that the resource manager identify a specific set of resources, or collect observation reports from these resources according to a specific location specification.

4.4 Event notification

In addition to direct observation that shows in the form of requests for observation reports sent to resources, D-RAJE allows observation to be delegated to parameterizable monitors. Indeed, The implementation of a simple – yet flexible – event model makes it possible to ask monitor objects to periodically observe some resources so as to be notified when specific events occur regarding the state of these resources.

The description of a set of interesting events is modelled by an *ObservationProfile* object which comprises a set of resource ids and an object implementing the *ObservationReportPattern* interface. Observation report patterns are to observation reports what resource patterns are to resources. Interface *ObservationReportPattern* is the root of a hierarchy of classes that serve as event filters. Each of these classes provides a boolean method *isMatchedBy* that implements a predicate on an obervation report passed as a parameter. A the top level of the hierarchy we have report patterns that are used to filter events according to resource types (for example method *IsMatchedby* of class *SocketPattern* only verifies that its parameter is of type *SocketReport*). Under this level, we have defined more precise patterns that covers usual events. For example, the class *LowFreeMemoryPattern* makes it possible to detect that the free memory attribute value of a memory report is under a given threshold. The hierarchy can be easily extended by defining specific patterns for other events as long as they each pertain to one resource.

Event publishing is performed by an object implementing the *ResourceMonitor* interface that handles a set of observation profiles. It is in charge of observing, at a given period, the resources cited in all its observation profiles and notifying the corresponding event sinks (an object implementing interface *Notifiable* is stored in each observation profile). Two classes currently implement the *ResourceMonitor* interface, offerring distinct monitoring schemes. In the first one, a resource monitor creates only one periodic observer (mainly a thread) that fetches possibly distant observation reports. Notification is thus performed locally. In the second scheme, periodic observers are dispatched close to resources and event notifications are transferred through the network. Event subscription is done by instanciating a monitor and passing it a set of observation profiles. Moreover, after a monitor has been created, its behaviour can be dynamically modified by changing its observation period or by adding/removing observation profiles.

Event notification is achieved by calling the *notify* method on the event sink object. The call holds a reference to an *ObservationEvent* object that contains a local timestamp as well as the information necessary to identify the origin of the event (a resource id) and the corresponding subscription (a reference to a resource monitor and an observation profile).

The following piece of code illustrates how a simple subscription is performed. In this case the user is interested in being notified whenever the free memory level of any node in the network goes either below 5 Mo or beyond 100 Mo. First the ids of all the memory objects are gathered (line 1); then two observation profiles are built, corresponding to the two kind of desired events (lines 2 and 3); and lastly the observation profiles are passed to a dispatched monitor constructor (line 4). The state of the memories will be observed locally with a period of 1000 ms. Notice that the object performing these calls is specified as the event sink for all the events generated (parameter *this* in lines 2 and 3).

```
// (1)
Set allMemIds =
    manager.getResourceIds(new MemoryPattern(), new GlobalSearch());
// (2)
ObservationProfile opLow =
    new ObservationProfile(allMemids,
                           new LowFreeMemoryPattern(5*Mo),
                           this);
// (3)
ObservationProfile opHigh=
    new ObservationProfile(allMemids,
                           new HighFreeMemoryPattern(100*Mo),
                           this);
// (4)
ResourceMonitor rm =
    new DispatchedResourceMonitor({opLow, opHigh},
                                  1000);
```

5 Implementation details

Obtaining in Java some information about hardware or system parts implies that the objects reifying these resources in D-RAJE interact with the underlying operating system in order to collect the pieces of information requested by the calling program. The implementation of the classes modelling system resources in D-RAJE relies on native C code, whose role is to get information from the OS. So far D-RAJE was only implemented over Linux; though, the architecture of D-RAJE was carefully designed so that it can be ported at low cost on a variety of operating systems.

The approach we have chosen for implementing conceptual resources is to change the definition and implementation of standard JDK classes when necessary. Only minor modifications were required in the source code of classes like *Socket*, *DatagramSocket*, and *File*. The problem was more complex for the class *Thread*, though, since our objective was to permit that the CPU time and the memory space consumed by each Java thread be observable at runtime. To achieve this goal the actual implementation of Java threads in the JVM (Kaffe 1.07) had to be altered significantly and the memory allocator had too be also modified. It is worth mentioning that the signatures of the methods defined originally in the JDK were left untouched, and the semantics of these methods was preserved as well.

Although performance was not our prior concern in the initial implementation, a set of preliminary measurements have been conducted, mainly to obtain the order of magnitude of observation timings, especially as far as distant access to resource information is concerned. A few partial results are presented in the following table. In this experiment, a client asks a server for an observation report concerning four different resources: the CPU, the physical memory, an Ethernet interface and a user-defined dummy resource that produces empty observation reports. The client and the server run on 2.5 GHz Pentium 4 machines (with Linux 2.4) linked by an Ethernet network (100 Mb/s and 10 Mb/s). The last line of the table reports analogous resource observations using RAJE, the version of D-RAJE that restricts observation to the local host. The results show that although performances vary according to the type of resource, timings remain acceptable. Several ways of improvement are clearly possible, namely by replacing RMI by a more efficient transmission mode for cooperation between resource managers and by optimizing the management of the internal data structures of D-RAJE.

	CPU	Memory	Eth. Int.	Dummy
D-RAJE 100 Mb/s	1.8 ms	1.5 ms	7.8 ms	1.4 ms
D-RAJE 10 Mb/s	2.5 ms	2.0 ms	8.5 ms	1.9 ms
RAJE	0.095 ms	0.095 ms	5.0 ms	0.002 ms

6 Conclusion

This paper has presented an overview of D-RAJE *(Distributed Resource-Aware Java Environment)*, an open, object-oriented Java middleware architecture that makes it possible to model and to monitor resources in a distributed environment. The types of resources considered goes beyond classical system resources such as memory or CPU. With D-RAJE, the user may model resources that are not directly associated with hardware devices. For example, applicative entities like a socket, a file, a RMI registry or a user thread may also be considered as resources.

D-RAJE permits the identification of resources and the consultation of their state across a distributed system. The objective is to provide homogeneous access to resource information. A distributed resource manager offers services to search for resources based on possibly complex selection criteria, and to obtain observation reports from the resources thus selected. Mechanisms for notification of events concerning resources' states are also available.

D-RAJE defines and implements an extensible framework that may be used to ease the development of adaptive, security-oriented, or QoS-oriented Java applications. It is not intended directly for end-users but could rather form a basis for higher level middleware. In this perspective, D-RAJE has been used in two projects that are being carried out in our laboratory. The first one is project Concerto which aims at allowing the deployment and the support of parallel adaptive software components on non-dedicated clusters of workstations [4]. Concerto uses the facilities offered by D-RAJE to permit the definition of dynamic adaptation strategies, based on the observation of resources of any kind. The extensibility of D-RAJE was exploited in project Concerto, as new types of resources were defined specifically for this project. JAMUS is another middleware architecture built on top of D-RAJE in order to experiment with the idea of resource contracting[10]. JAMUS is a platform that supports the deployment of so-called "untrusted" software components, provided that these components can

specify their requirements regarding resource utilisation in both qualitative and quantitative terms (eg acces permissions and access quotas). Emphasis is put on providing a safe and guaranteed runtime environment for such components. Resource control in JAMUS is based on a contractual approach. Whenever a software component applies for being deployed on the platform, it must specify explicitly what resources it will need at runtime, and in what conditions. Thanks to the resource observation services of D-RAJE, JAMUS can provide some level of quality of service regarding resource availability. It also provides components with a relatively safe runtime environment, since no component can access or monopolise resources to the detriment of other components.

The D-RAJE platform is still under construction. In a near future, we plan to augment the set of resources modelled (laptop battery, USB devices,...) and to extend the event model so that compound events, pertaining to several resources at the same time, could be handled.

Aknowledment

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

References

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *4th Symposium on Operating Systems Design and Implementation*, October 2000.

[2] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *USENIX Annual Technical Conference*, June 2000.

[3] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. The Legion Resource Management System. In *5th Workshop on Job Scheduling Strategies for Parallel Processing, IPDPS'99*, April 1999.

[4] L. Courtrai, F. Guidec, N. Le Sommer, and Y. Mahéo. Resource Management for Parallel Adaptive Components. In *5th Int. Workshop on Java for Parallel and Distributed Computing, IPDPS'03*, Nice, France, April 2003.

[5] G. Czajkowski and T. von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.

[6] 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.

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

[8] 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, August 2000.

[9] K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Sofware – Practice and Experience*, 32(2):135–164, February 2002.

[10] N. Le Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *1st Int. IFIP/ACM Working Conference on Component Deployment*, Berlin, Germany, June 2002.

[11] M. Neary, A. Phipps, S. Richman, and P. Capello. Javalin 2.0: Java-based Parallel Computing on the Internet. In *European Parallel Computing Conference (Euro-Par'2000)*, Munich, Germany, August 2000.

[12] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *7th IEEE Int. Symposium on High Performance Distributed Computing*, Chicago, USA, July 1998.