Programming, Composing, Deploying for the Grid

Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes,
Fabrice Huet, Matthieu Morel, and Romain Quilici

OASIS - Joint Project CNRS / INRIA / University of Nice Sophia-Antipolis
INRIA - 2004 route des Lucioles - B.P. 93 - 06902 Valbonne Cedex, France
FirstName.LastName@sophia.inria.fr

Abstract. Grids raise new challenges in the following way: heterogeneity of underlying machines/networks and runtime environments (types and performance characteristics), not a single administrative domain, versatility. So the need to have appropriate programming and runtime solutions in order to write, deploy then execute applications on such heterogeneous distributed hardware in an effective and efficient manner. We propose in this article a solution to those challenges which takes the form of a programming and deployment framework featuring parallel, mobile, secure and distributed objects and components.
Keywords: Distributed objects, components, mobility, monitoring, deployment, security, mapping.

1 Introduction

1.1 Motivation

In this article, we present a contribution to the problem of software reuse, integration and deployment for parallel and distributed computing. Our approach takes the form of a programming and deployment framework featuring parallel, mobile, secure and distributed objects and components. We especially target Grid computing, but our approach also applies to application domains such as mobile and ubiquitous distributed computing on the Internet (where high performance, high availability, ease of use, etc., are of importance).

Below are the main current problems raised by grid computing that we have identified, and for which we provide some solutions. For Grid applications development, there is a need to smoothly, seamlessly and dynamically integrate and deploy autonomous software, and for this to provide a *glue* in the form of a software bus. Additionally, complexification of distributed applications and commodity of resources through grids are making the tasks of deploying those applications harder. So, there is a clear need for standard tools allowing versatile deployment and analysis of distributed applications. Grid applications must be able to cope with large variations in deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. As a consequence, the security should not be tied up in the application code, but rather

easily configurable in a flexible, and abstract manner. Moreover, any large scale Grid application using hundreds or thousands of nodes will have to cope with migration of computations, for the sake of load balancing, change in resource availability, or just node failures.

We propose programming concepts, methodologies, and a framework for building meta-computing applications, that we think are well adapted to the hierarchical, highly distributed, highly heterogeneous nature of grid-computing. The framework is called *ProActive*, a Java-based middleware (programming model and environment) for object and component oriented parallel, mobile and distributed computing. As this article will show, *ProActive* is relevant for grid computing due to its secure deployment and monitoring aspects [1, 2], its efficient and typed collective communications [3], and component-based programming facilities [4] thanks to an implementation of the Fractal component model [5, 6], taking advantage of its hierarchical approach to component programming.

1.2 Context

Grid programmers may be categorized into three groups, such as defined in [7]: the first group are end users who program pre-packaged Grid applications by using a simple graphical or Web interface; the second group are those that know how to build Grid applications by composing them from existing application "components", for instance by programming (using scripting or compiled languages); the third group consists of the developers that build the individual components. Providing the user view of the Grid can also be seen as a two-levels programming model [8]: the second level is the integration of distributed components (developped at the first level), together into a complete *executable*.

In this context, the component model we propose addresses the second group of programmers; but we also address the third group by proposing a deployment and object-oriented programming model for *autonomous* grid-aware distributed software that may further be integrated if needed.

In the context of object oriented computing for grid, for which security is a concern, works such as Legion [9, 10] also provide an integrated approach like we do. But the next generation of programming models for wide area distributed computing is aimed at further enforcing code reuse and simplifying the developer's and integrator's task, by applying the component oriented methodology. We share the goal of providing a component-based high-performance computing solution with several projects such as: CCA [7] with the CCAT/XCAT toolkit [11] and Ccaffeine framework, Parallel CORBA objects [12] and GridCCM [13]. But, to our knowledge, what we propose is the first framework featuring *hierarchical* distributed components. This should clearly help in mastering the complexity of composition, deployment, re-usability required when programming and running large-scale distributed applications.

1.3 Plan

The organization of the paper is as follows: first we describe the parallel and distributed programming model we advocate for developing autonomous grid-aware software. Second, we define the concept of hierarchical, parallel, and distributed components yielding the concept of *Grid components*. The third part of the paper deals with concepts and tools useful during the life-cycle of a Grid application: deployment that moreover might need to be secure, visualization and monitoring of a Grid application and its associated runtime support (e.g. Java Virtual Machines), re-deployment by moving running activities and by rebuilding components.

2 Programming Distributed Mobile Objects

2.1 Motivation

As grid computing is just one particular instance of the distributed computing arena, our claim is that proposed programming models and tools should not drastically depart from traditional distributed computing.

We present here the parallel and distributed conceptual programming model and at the same time, one of the many implementations we have done, called *ProActive*, which is in Java, besides others done with Eiffel and C++, resp. called *Eiffel//* and *C++//* [14, 15]. The choice of the Java language is fundamental in order to hide heterogeneity of runtime supports, while the performance penalty is not too high compared with native code implementations (c.f. Java Grande benchmarks for instance [16]).

As *ProActive* is built on top of the Java standard APIs, mainly Java RMI and the Reflection APIs, it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified virtual machine. Additionally, the Java platform provides dynamic code loading facilities, very useful for tackling with complex deployment scenarios.

2.2 Base Model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [17]. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee. All of this semantics is built using meta programming techniques, which provide transparency and

the ground for adaptation of non-functional features of active objects to various needs.

We have deliberately chosen not to use an explicit message-passing based approach: we aim at enforcing code reuse by applying the remote method invocation pattern, instead of explicit message-passing.

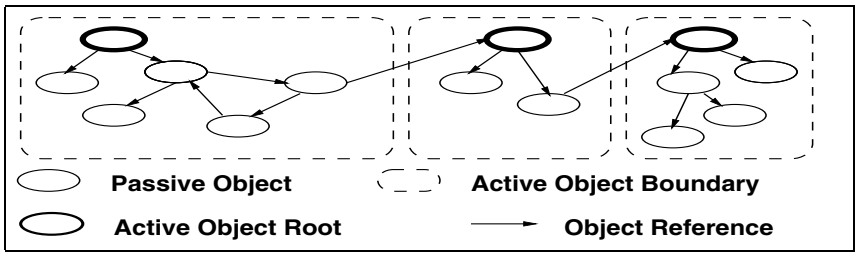


Fig. 1. A typical object graph with active objects

Another extra service (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For the sake of generality and dynamicity, creations of remotely accessible objects are triggered entirely at runtime by the application; nevertheless, active objects previously created and registered within another application may be accessed by the application, by first acquiring them with a lookup operation.

For that reason, JVMs need to be identified and added a few services. *Nodes* provide those extra capabilities : a *node* is an object defined in the model whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance `rmi://lo.inria.fr/Node`.

Let us consider a standard Java class `A`:

```
class A {
  public A() {}
  public void foo (...) {...}
  public V bar (...) {...}
    ...
}
```

The instruction:

```
A a = (A) ProActive.newActive("A",params,"//lo.inria.fr/Node");
```

creates a new active object of type `A` on the JVM identified with `Node`. It assumes that `Node` (i.e. the JVM) has been already deployed (see below and also section

4). Note that an active object can also be bound dynamically to a node as the result of a migration. Further, all calls to that remote object will be asynchronous, and subject to the *wait-by-necessity*:

```
a.foo (...);     // Asynchronous call
v = a.bar (...); // Asynchronous call
 ...
v.f (...);       // Wait-by-necessity: wait until v gets its value
```

2.3 Mobility

ProActive provides a way to move an active object from any Java virtual machine to any other one. This feature is accessible through a simple `migrateTo(...)` primitive (see Table 1).

	Migration towards:
`migrateTo (URL)`	a VM identified by a URL
`migrateTo (Objet)`	the location of another Active Object

Table 1. Migration primitives in *ProActive*

The primitive is `static`, and as such always triggers the migration of the current active object and all its passive objects. However, as the method is `public`, other, possibly remote, active objects can trigger the migration; indeed, the primitive implements what is usually called *weak migration*. The code in Example 2 presents such a mobile active object.

```
public class SimpleAgent implements Serializable {
  public void moveToHost(String t) {
      ProActive.migrateTo(t);
  }
  public void joinFriend(Object friend) {
      ProActive.migrateTo(friend);
  }
  public ReturnType foo(CallType p) {
      ...
  }
}
```

Fig. 2. SimpleAgent

In order to ensure the working of an application in the presence of migration, we provide three mechanism to maintain communication with mobile objects.

The first one relies on a location sever which keeps track of the mobile objects in the system. When needed, the server is queried to obtain an up-to-date reference to an active object. After migrating, an object updates its new location.

The second one uses a fully decentralized technique known as *forwarders* [18]. When leaving a site, an active object leaves a special object called a forwarder which points to its new location. Upon receiving a message, a forwarder simply passes it to the object (or another forwarder).

The third one is an original scheme based on a mix between forwarding an location server which provides both performance and fault tolerance.

2.4 Group Communications

The group communication mechanism of *ProActive* achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies.

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remain typed. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.

Here is an example of a typical group creation, based on the standard Java class `A` presented above:

```
  // A group of type "A"  and its 2 members are created at once
  // on the nodes directly specified,
  //  parameters are specified in params,
Object[][] params = {{...}, {...}};
Node[] nodes = {..., ...};
A ag = (A) ProActiveGroup.newActiveGroup("A", params, nodes);
```

Elements can be included into a typed group only if their class equals or extends the class specified at the group creation. Note that we do allow and handle *polymorphic* groups. For example, an object of class `B` (`B` extending `A`) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

A method invocation on a group has a syntax similar to that of a standard method invocation:

```
ag.foo(...); // A group communication
```

Such a call is asynchronously propagated to all members of the group using multi-threading. Like in the *ProActive* basic model, a method call on a group is non-blocking and provides a transparent future object to collect the results. A method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be a *ProActive* call and if the member is a standard Java object, the method call will be a standard Java

method call (within the same JVM). The parameters of the invoked method are broadcasted to all the members of the group.

An important specificity of the group mechanism is: the *result* of a typed group communication *is also a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited the caller blocks, but as soon as one reply arrives in the result group, the method call on this result is executed. For instance in:

```
V vg = ag.bar(); // A method call on a group, returning a result
                 // vg is a typed group of "V"
vg.f();          // This is also a collective operation
```

a new `f()` method call is automatically triggered as soon as a reply from the call `ag.bar()` comes back in the group `vg` (dynamically formed). The instruction `vg.f()` completes when `f()` has been called on all members.

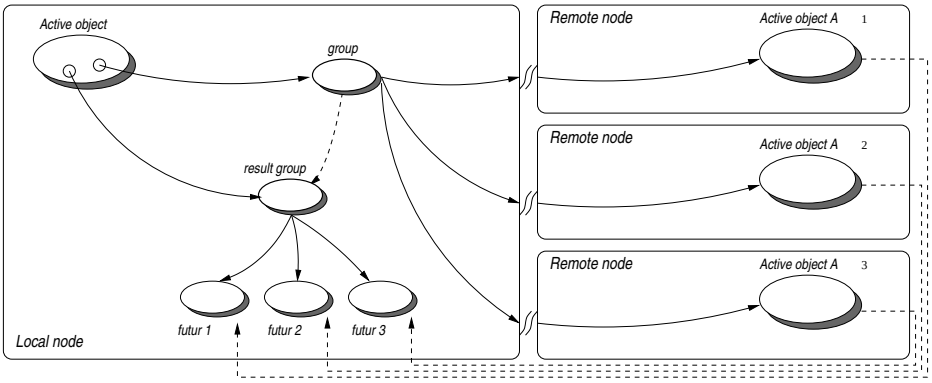


Fig. 3. Execution of an asynchronous and remote method call on group with dynamic generation of a result group

Other features are available regarding group communications: parameter dispatching using groups (through the definition of *scatter* groups), hierarchical groups, dynamic group manipulation (`add`, `remove` of members), group synchronization and barriers (`waitOne`, `waitAll`, `waitAndGet`); see [3] for further details and implementation techniques.

2.5 Abstracting Away from the Mapping of Active Objects to JVMs: Virtual Nodes

Active objects will eventually be deployed on very heterogeneous environments where security policies may differ from place to place, where computing and

communication performances may vary from one host to the other, etc. As such, the effective locations of active objects must not be tied in the source code.

A first principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols,

the goal being to deploy any application anywhere without changing the source code. For instance, we must be able to use various protocols, `rsh, ssh, Globus, LSF,` etc., for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as `RMIregistry, Jini, Globus, LDAP, UDDI,` etc. Therefore, we see that the creation, registration and discovery of resources has to be done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. The description should indicate the various parallel or distributed entities in the program or in the component. As we are in a (object-oriented) message passing model, to some extend, this description indicates the maximum number of address spaces. For instance, an application that is designed to use three interactive visualization nodes, a node to capture input from a physic experiment, and a simulation engine designed to run on a cluster of machines should somewhere clearly advertise this information.

Now, one should note that the abstract description of an application and the way to deploy it are not independent piece of information. In the example just above, if there is a simulation engine, it might register in a specific registry protocol, and if so, the other entities of the computation might have to use that lookup protocol to bind to the engine. Moreover, one part of the program can just lookup for the engine (assuming it is started independently), or explicitly create the engine itself.

To summarize, in order to abstract away the underlying execution platform, and to allow a *source-independent deployment*, a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

Besides the principles above, we want to eliminate as much as possible the use of scripting languages, that can sometimes become even more complex than application code. Instead, we are seeking a solution with XML descriptors, XML editor tools, interactive ad-hoc environments to produce, compose, and activate descriptors (see section 4).

To reach that goal, the programming model relies on the specific notion of **Virtual Nodes** (VNs):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN is defined and configured in a deployment descriptor (XML) (see section 4 for further details),
- a VN, after activation, is mapped to one or to a set of *actual* ProActive *Nodes*.

Of course, distributed entities (active objects), are created on Nodes, not on Virtual Nodes. There is a strong need for both Nodes and *Virtual Nodes*. *Virtual Nodes* are a much richer abstraction, as they provide mechanisms such as *set* or *cyclic mapping*. Another key aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical if we want to get at once machines from a cluster of PCs managed through Globus or LSF. It is even more critical in a Grid application, when trying to achieve the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a deployment descriptor (see section 4):

- the mapping of VNs to Nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup VNs.

```
ProActiveDescriptor pad =
            ProActive.getProActiveDescriptor(String xmlFileLocation);
//---- Returns a ProActiveDescriptor object from the xml file
VirtualNode dispatcher = pad.getVirtualNode("Dispatcher");
//---- Returns the VirtualNode Dispatcher described
//     in the xml file as a java object
dispatcher.activate()
// --- Activates the VirtualNode
Node node = dispatcher.getNode();
//-----Returns the first node available among nodes mapped
//     to the VirtualNode
C3DDispatcher c3dDispatcher = newActive("C3DDispatcher", param, node);
.........................
```

Fig. 4. Example of a *ProActive* source code for descriptor-based mapping

Now, within the source code, the programmer can manage the creation of active objects without relying on machine names and protocols. For instance, the piece of code given in Figure 4 will allow to create an active object onto the Virtual Node `Dispatcher`. The Nodes (JVMs) associated in a descriptor file with a given VN are started (or acquired) only upon activation of a VN mapping (`dispatcher.activate()` in the Figure 4).

3 Composing

3.1 Motivation

The aim of our work around components is to combine the benefits of a component model with the features of *ProActive*. The resulting components, that we call "Grid components", are recursively formed of either sequential, parallel and/or distributed sub-components, that may wrap legacy code if needed, and that may be deployed but further reconfigured and moved – for example to tackle fault-tolerance, load-balancing or adaptability to changing environmental conditions.

Here is a typical scenario illustrating the usefulness of our work. Consider complex grid software formed of several services, say of other software (a parallel and distributed solver, a graphical 3D renderer, etc). The design of this grid software is highly simplified if it can be considered as a hierarchical composition (recursive assembly and binding): the solver is itself a component composed of several components, each one encompassing a piece of the computation. The whole software is seen as a single component formed of the solver and the renderer. From the outside, the usage of this software is as simple as invoking a functional service of a component (e.g. call *solve-and-render*). Once deployed and running on a grid, assume that due to load balancing purposes, this software needs to be relocated. Some of the ongoing computations may just be moved (the solver for instance); others depending on specific peripherals that may not be present at the new location (the renderer for instance) may be terminated and replaced by a new instance adapted to the target environment and offering the same service. As the solver is itself a hierarchical component formed of several sub-components, each encompassing an activity, we trigger the migration of the solver as a whole, without having to explicitly move each of its sub-components, while references towards mobile components remain valid. Eventually, once the new graphical renderer is launched, we re-bind the software, so as it now uses this new configuration.

3.2 Component Model

Observing the works done so far on component software, including standardized industrial component models, such as CCM, EJB or COM, some researchers concluded that there was still missing an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. They

consequently introduced a new model[5], based on the concepts of encapsulation (components are black boxes), composition (the model is hierarchical), sharing[1], life-cycle (a component lives through different phases), activities, control (this allows the management of non-functional properties of the components), and dynamicity (this allows reconfiguration). This model is named Fractal.

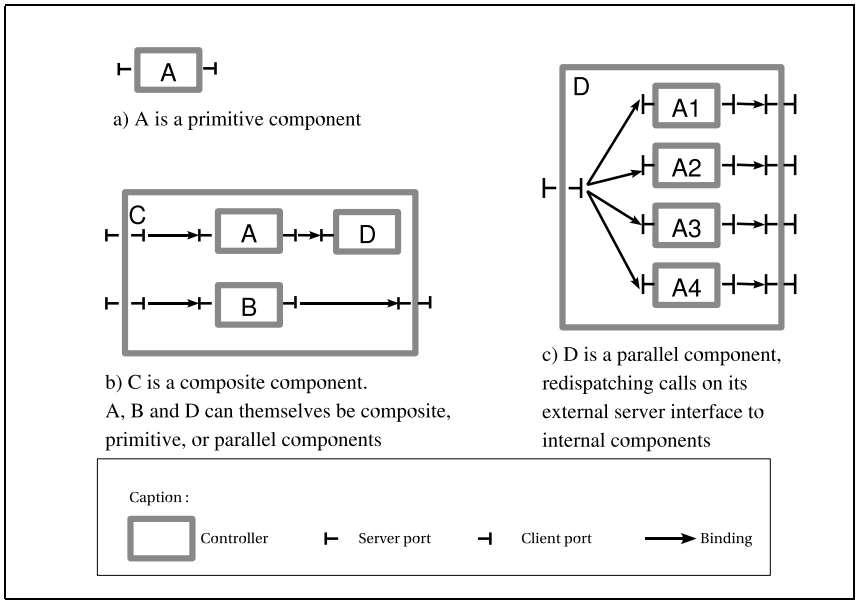


Fig. 5. The 3 types of components

The Fractal model is somewhat inspired from biological cells, i.e. plasma surrounded by membranes. In other words, a component is formed out of two parts: a *content*, and a set of *controllers*. The content can be recursive, as a component can contain other components: the model is hierarchical. The controllers provide introspection capabilities for monitoring and exercising control over the execution of the components. A component interacts with its environment (notably, other components) through well-defined *interfaces*. These interfaces can be either client or server, and are interconnected using *bindings* (see fig. 5).

Fractal is a component model conceived to be simple but extensible. It provides an API in Java, and offers a reference implementation called Julia. Unfortunately, Julia is not based on a distributed communication protocol (although there exists a Jonathan personality, i.e. a set of RMI Fractal components), thus hindering the building of systems with distributed components.

[1] sharing is currently not supported in the *ProActive* implementation

Besides, *ProActive* offers many features, such as distribution, asynchronism, mobility or security, that would be of interest for Fractal components.

We therefore decided to write a new implementation of the Fractal API based on *ProActive*, that would benefit from both sides, and that would ease the construction of distributed and complex systems.

3.3 *ProActive* Components

A *ProActive* component has to be parallelizable and distributable as we aim at building grid-enabled applications by hierarchical composition; componentization acts as a glue to couple codes that may be parallel and distributed codes requiring high performance computing resources. Hence, components should be able to encompass more than one activity and be deployed on parallel and distributed infrastructures. Such requirements for a component are summarized by the concept we have named *Grid Component*.

Figure 5 summarizes the three different cases for the structure of a Grid component as we have defined it. For a composite built up as a collection of components providing common services (Figure 5.c), *group communications* (see 2.4) are essential for ease of programming and efficiency. Because we target high performance grid computing, it is also very important to efficiently implement point-to-point and group method invocations, to manage the deployment complexity of components distributed all over the Grid and to possibly debug, monitor and reconfigure the running components.

A synthetic definition of a *ProActive* component is the following :

- It is formed from one (or several) Active Object(s), executing on one (or several) JVM(s)
- It provides a set of server ports (Java Interfaces)
- It possibly defines a set of client ports (Java attributes if the component is primitive)
- It can be of three different types :
  1. primitive : defined with Java code implementing provided server interfaces, and specifying the mechanism of client bindings.
  2. composite : containing other components.
  3. parallel : also a composite, but re-dispatching calls to its external server interfaces towards its inner components.
- It communicates with other components through 1-to-1 or group communications.

A *ProActive* component can be configured using :

- an XML descriptor (defining use/provide ports, containment and bindings in an Architecture Description Language style)
- the notion of virtual node, capturing the deployment capacities and needs

Finally, we are currently working on the design of specialized components encapsulating legacy parallel code (usually Fortran-MPI or C-MPI). This way, *ProActive* will allow transparent collaboration between such legacy applications and any other Grid component.

3.4 Example

We hereby show an example of how a distributed component system could be built using our component model implementation. It relates to the scenario exposed in section 3.1.

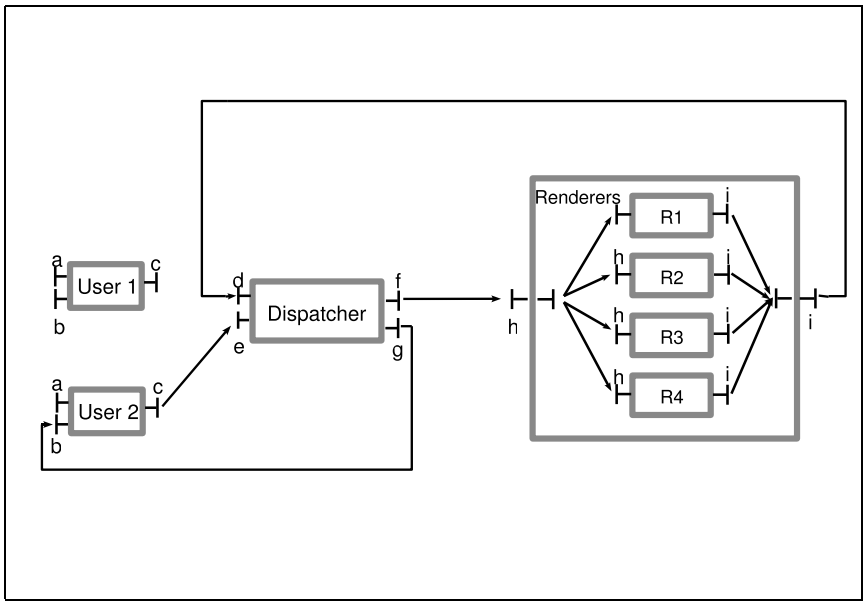


Fig. 6. A simplified representation of the C3D component model

C3D, an existing application, is both a collaborative application and a distributed raytracer: users can interact through messaging and voting facilities in order to choose a 3D scene that is rendered using a set of distributed rendering engines working in parallel. This application is particularly suitable for component programming, as we can distinguish individual software entities and we can abstract client and server interfaces from these entities. The resulting component system is shown in figure 6 : users interact with the dispatcher component, can ask for the scene motion, and can see the evolution of the ray-tracing. The dispatcher delegates the calculation of the scene to a parallel component (renderers). This parallel component contains a set of rendering engines (R1, R2, R3), and distributes calculation units to these rendering engines thanks to the

group communication API (scatter feature, see 2.4). The results are then forwarded (from the client interface `i` of the renderers component) as a call-back to the dispatcher (server interface `d`), and later to the users (from client interface `g` of the dispatcher to server interface `b` of the clients). These relations result in cyclic composition of the components. During the execution, users (for instance user1 represented on the figure) can dynamically connect to the dispatcher and interact with the program. Another dynamical facility is the connection of new rendering engine components at runtime : to speedup the calculations, if the initial configuration does not perform fast enough, new rendering engines can be added along with R1, R2, R3. Besides, components being active objects, they can also migrate for load-balancing or change of display purposes, either programmatically or interactively using tools such as IC2D (see 4.3). Interaction between users, like votes, is not represented here.

There are two ways of configuring and instantiating component systems : either programmatically, or using an architecture description language (ADL). The ADL can help a lot, as it automates the instantiation, the deployment, the assembly and the binding of the components. The following examples correspond to the configuration of the C3D application. The ADL is composed of two main sections. The first section defines the types of the components (User-Type, Dispatcher-Type and Renderer-Type), in other words the services the components offer and the services they require :

```
<types>
    <component-type name="User-Type">
        <provides>
            <interface name="a" signature="package.UserInput"/>
            <interface name="b" signature="package.SceneUpdate"/>
        </provides>
        <requires>
            <interface name="c" signature="package.UserSceneModification"/>
        </requires>
    </component-type>
    <component-type name="Dispatcher-Type">
        <provides>
            <interface name="d" signature="package.UserSceneModification"/>
            <interface name="e" signature="package.CalculationResult"/>
        </provides>
        <requires>
            <interface name="f" signature="package.CalculateScene"/>
            <interface name="g" signature="package.SceneUpdate"/>
        </requires>
    </component-type>
    <component-type name="Renderer-Type">
        <provides>
            <interface name="h" signature="package.Rendering"/>
        </provides>
        <requires>
            <interface name="i" signature="package.CalculationResult"/>
```

```
        </requires>
    </component-type>
</types>
```

The second section defines the instances of the components, the assembly of components into composites, and the bindings between components :

```
<components>
    <primitive-component implementation="package.User"
                    name="user1" type="User-Type"
                    virtualNode="UserVN"/>
    <primitive-component implementation="package.User"
                    name="user2" type="User-Type"
                    virtualNode="UserVN"/>
    <primitive-component implementation="package.Dispatcher"
                    name="dispatcher" type="Dispatcher-Type"
                    virtualNode="DispatcherVN"/>
    <parallel-component name="parallel-renderers"
                    type="Renderer-Type"
                    virtualNode="parallel-renderers-VN">
        <components>
            <primitive-component implementation="package.Renderer"
                    name="renderer" type="Renderer-Type"
                    virtualNode="renderers-VN"/>
            <!-- the actual number of renderer instances
            depends upon the mapping of the virtual node -->
        </components>
        <!-- bindings are automatically performed
        inside parallel components -->
    </parallel-component>
</components>
<bindings>
    <binding client="dispatcher.c" server="parallel-renderers.c"/>
    <binding client="renderers.r" server="dispatcher.r"/>
    <binding client="user1.i" server="dispatcher.i"/>
    <binding client="dispatcher.g" server="user2.b"/>
    <!-- bindings to clients can also be performed dynamically
    as they appear once the application is started
    and ready to receive input operations -->
</bindings>
```

Bindings connect components at each level of the hierarchy, and are performed automatically inside parallel components. The primitive components contain functional code from the class specified in the implementation attribute.

Each component also exhibits a "virtual node" property : the design of the component architecture is decoupled from the deployment (see 4.2) of the components. This way, the same component system can be deployed on different computer infrastructures (LAN, cluster, Grid).

In conclusion, the benefits of the componentization of the C3D application are – at least – threefold. First, the application is easier to understand and to

configure. Second, the application is more evolutive: for instance, as the rendering calculations are encapsulated in components, one could improve the rendering algorithm, create new rendering engine components and easily replace the old components with the new ones. Third, the application is easier to deploy, thanks to the mapping of the components onto virtual nodes.

4 Deploying, Monitoring

4.1 Motivation

Increasing complexity of distributed applications and commodity of resources through grids are making the tasks of deploying those applications harder. There is a clear need for standard tools allowing versatile deployment and analysis of distributed applications. We present here concepts for the deployment and monitoring, and their implementation as effective tools integrated within the *ProActive* framework. If libraries for parallel and distributed application development exist (RMI in Java, jmpi [19] for MPI programming, etc.) there is no standard yet for the deployment of such applications. The deployment is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers, clusters or grids. The commoditization of resources through grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

Questions such as "are the distributed entities correctly created ?", "do the communications among such entities correctly execute ?", "where is a given mobile entity actually located ?", etc. are usually left unanswered. Moreover, there is usually no mean to dynamically modify the execution environment once the application is started. Grid programming is about deploying processes (activities) on various machines. In the end, the security policy that must be ensured for those processes depends upon many factors: first of all, the application policy that is needed, but also, the machine locations, the security policies of their administrative domain, and the network being used to reach those machines.

Clearly said, the management of the mapping of processes (such as JVMs, PVM or MPI daemons) onto hosts, the deployment of activities onto those processes have generally to be explicitly taken into account, in a static way, sometimes inside the application, sometimes through scripts. The application cannot be seamlessly deployed on different runtime environments.

To solve those critical problems, the quite classical and somehow ideal solutions we propose follow 4 steps:

1. abstract away from the hardware and software runtime configuration by introducing and manipulating in the program virtual processes where the activities of the application will be subsequently deployed,
2. provide external information regarding all real processes that must be launched and the way to do it (it can be through remote shells or job submission to clusters or grids), and define the mapping of virtual processes onto real processes,

3. provide a mean to visualize, complete or modify the deployment once the application has started,
4. provide an infrastructure where Grid security is expressed outside the application code, outside the firewall of security domains, and in both cases in a high-level and flexible language.

4.2 Deployment Descriptors

We solve the two first steps by introducing XML-based descriptors able to describe activities and their mapping onto processes. Deployment descriptors allow to describe: (1) virtual nodes, entities manipulated in the source code, representing containers of activities, (2) Java virtual machines where the activities will run and the way to launch or find them, (3) the mapping between the virtual nodes and the JVMs. The deployment of the activities is consequently separated from the code; one can decide to deploy the application on different hosts just by adapting the deployment descriptor, without any change to the source code.

Descriptors are structured as follows:

```
virtual nodes
        definition
        acquisition
deployment
        security
        register
        lookup
        mapping
        jvms
infrastructure
```

Virtual Nodes As previously stated (see 2.5), a virtual node is a mean to define a mapping between a conceptual architecture and one or several nodes (JVMs), and its usage in the source code of a program has been given on figure 4.

The names of the virtual nodes in the source code has to correspond to the names of the virtual nodes defined in the first section. There are two ways of using virtual nodes. The first way is to name them (and further explicitly describe them):

```
<virtualNodesDefinition>
    <virtualNode name="User"/>
</virtualNodesDefinition>
```

The second way is to acquire virtual nodes already deployed by another application:

```
<virtualNodesAcquisition>
    <virtualNode name="Dispatcher"/>
</virtualNodesAcquisition>
```

Deployment The deployment section defines the correspondence, or mapping, between the virtual nodes in the first section, and the processes they actually create.

The security section allows for the inclusion of security policies in the deployment (see section 4.4):

```
<security file="URL">
```

The register section allows for the registration of virtual nodes in a registry such as `RMIRegistry` or `JINI lookup service`:

```
<register virtualNode="Dispatcher" protocol="rmi">
```

This way, the virtual node "Dispatcher", as well as all the JVMs it is mapped to, will be accessible by another application through the rmi registry.

Symmetrically, descriptors provide the ability to acquire a virtual node already deployed by another application, and defined as acquirable in the first section:

```
<lookup virtualNode="Dispatcher" host="machineZ" protocol="rmi"/>
```

The mapping section helps defining:

- a one virtual node to one JVM mapping:

```
<map virtualNode="User1">
    <jvmSet>
        <currentJvm protocol="rmi"/>
        <!-- currentJvm is the Jvm of the file parsing process -->
    </jvmSet>
</map>
```

- a one virtual node to a set of JVMs mapping:

```
<map virtualNode="Renderer">
    <jvmSet>
        <vmName value=Jvm1/>
        <vmName value=Jvm2/>
        <vmName value=Jvm3/>
        ...
    </jvmSet>
</map>
```

- the collocation of virtual nodes, when two virtual nodes have a common mapping on a JVM.

Virtual nodes represent sets of JVMs, and these JVMs can be remotely created and referenced using standard Grid protocols. Each JVM is associated with a creation process, that is named here but fully defined in the "infrastructure" section. For example, here is an example where Jvm1 will be created through a Globus process that is only named here, but defined in the infrastructure section:

```
<jvms>
    <jvm name="Jvm1">
        <acquisition method="rmi"/>
        <creation>
            <processReference refid="GlobusProcess"/>
        </creation>
    </jvm>
    ...
</jvms>
```

Infrastructure The infrastructure section explicitly defines which Grid protocols (and associated *ProActive* classes) to use in order to create remote JVMs.

The remote creation of a JVM implies two steps: first, the connection to a remote host, second, the actual creation of the JVM.

Let us start with the second step. Once connected to the remote host, the JVM can be created using the `JVMNodeProcess` class:

```
<processDefinition id="jvmProcess">
    <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
<processReference refid="localJvmCreation"/>
```

The first step, the connection process, can itself invoke other processes. For example, the connection to LSF hosts requires beforehand a ssh connection to the frontal of the cluster, then a bSub command to reach the hosts inside the cluster. When connected, the previously defined `localJvmCreation` process is called:

```
<processDefinition id="bsubInriaCluster">
    <bsubProcess
        class="org.objectweb.proactive.core.process.lsf.LSFBSubProcess">
            <processReference refid="localJvmCreation"/>
            <bsubOption>
                <processor>20</processor>
            </bsubOption>
    </bsubProcess>
</processDefinition>
<processDefinition id="sshProcess">
    <sshProcess
        class="org.objectweb.proactive.core.process.ssh.SSHProcess"
        hostname="sea.inria.fr">
        <processReference refid="bsubInriaCluster"/>
    </sshProcess>
</processDefinition>
```

Other protocols are supported, including : rsh, rlogin, ssh, Globus, PBS. Here is an example using Globus:

```
<processDefinition id="globusProcess">
    <globusProcess
        class="org.objectweb.proactive.core.process.globus.GlobusProcess"
        hostname="globus.inria.fr">
        <processReference refid="localJvmCreation"/>
        <globusOption>
            <count>15</count>
        </globusOption>
    </globusProcess>
</processDefinition>
```

More information about the descriptors and how to use them is given in [20].

Deployment of Components The ADL used for describing component systems associates each component with a virtual node (see 3.4). A component system can be deployed using any deployment descriptor, provided the virtual node names match. The parallel components take advantage of the deployment descriptor in another way. In the example of section 3.4, consider the parallel component named "renderers". It only defines one inner component "renderer", and this inner component is associated to the virtual node "renderers-VN". If "renderers-VN" is mapped onto a single JVM A, only one instance of the renderer will be created, on the JVM A. But if this "renderers-VN" is actually mapped onto a set of JVMs, one instance of the renderer will be created on each of these JVMs. This allows for large scale parallelization in a transparent manner.

4.3 Interactive Tools

We solve the third step mentioned in section 4.1 by having a monitoring application: *IC2D* (Interactive Control and Debugging of Distribution). It is a graphical environment for monitoring and steering distributed *ProActive* applications.

Monitoring the Infrastructure and the Mapping of Activities Once a *ProActive* application is running, *IC2D* enables the user to graphically visualize fundamental distributed aspects such as topology and communications (see figure 7).

It also allows the user to control and modify the execution (e.g. the mapping of activities onto real processes, i.e. JVMs, either upon creation or upon migration. Indeed, it provides a way to interactively **drag-and-drop any running active object** to move it to any node displayed by *IC2D* (see figure 8). This is a useful feature in order to react to load unbalance, to expected unavailability of a host (especially useful in the context of a *desktop grid*), and more importantly in order to help implementing the concept of a *pervasive grid*: mobile users need

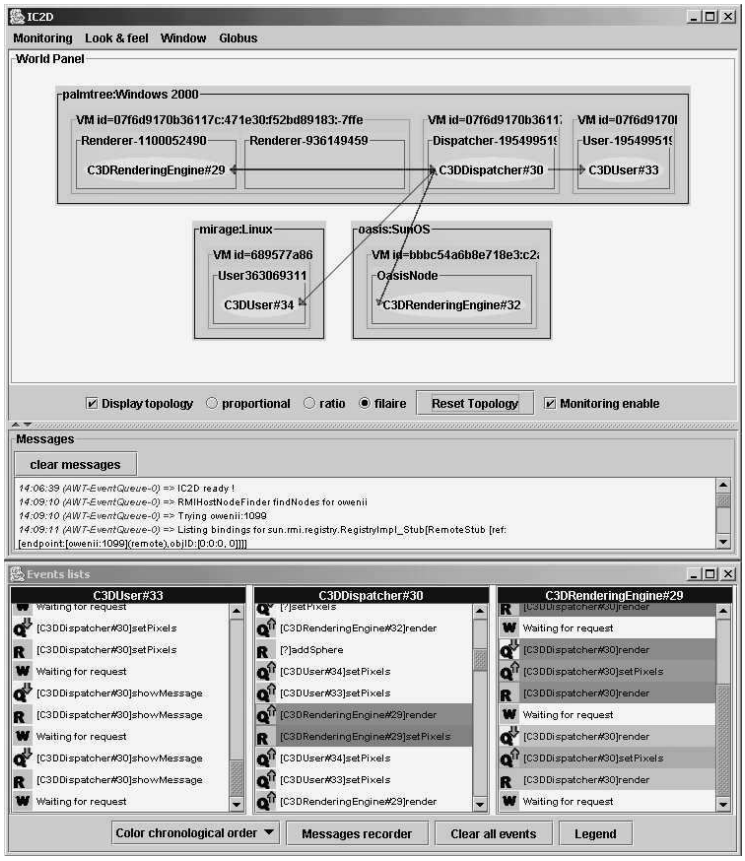


Fig. 7. General view of what *IC2D* displays when an application is running

to move the front-end active objects attached to the on-going grid computations they have launched, on their various computing devices so as to maintain their grid connectivity.

Moreover, it is possible to trigger the activation of new JVMs (see figure 9) adding dynamicity in the configuration and deployment.

Interactive Dynamic Assembly and Deployment of Components *IC2D compose* and *IC2D deploy* are two new interactive features we are planning to add to the *IC2D* environment.

The idea is to enable an integrator to graphically describe an application: the components, the inner components, and their respective bindings. The outcome of the usage of *IC2D compose* would be an automatically generated ADL. At this specific point, we need to provide the integrator with several solutions for **composing virtual nodes**. Indeed, as each *ProActive* component is attached to one virtual node, what about the virtual node of the composite component

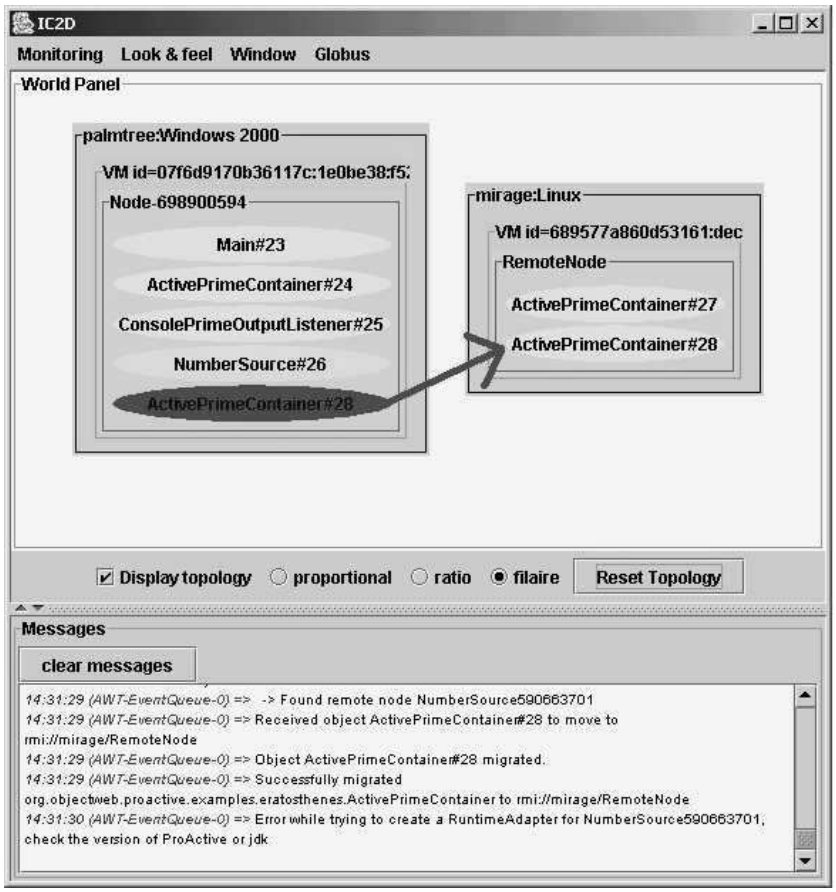


Fig. 8. Drag-and-drop migration of an active object

that results from the composition of several sub-components ? Should the resulting component still be deployed on the different virtual nodes of its inner components; or, on the contrary, should those virtual nodes be merged into one single virtual node attached to the composite ? The decision is grounded on the fact that merging virtual nodes is an application-oriented way to enforce the collocation of components.

Besides, *IC2D deploy* will be an additional tool that graphically would enable to trigger the deployment then the starting of an application based upon its ADL; this of course requires the complementary usage of a deployment descriptor attached to this application, so as to first launch the required infrastructure. Once the application has been deployed, *IC2D deploy* would enable to dynamically and graphically manage the life-cycle of the components (start, stop), and interface with *IC2D compose* so as to allow the user modify their bindings and

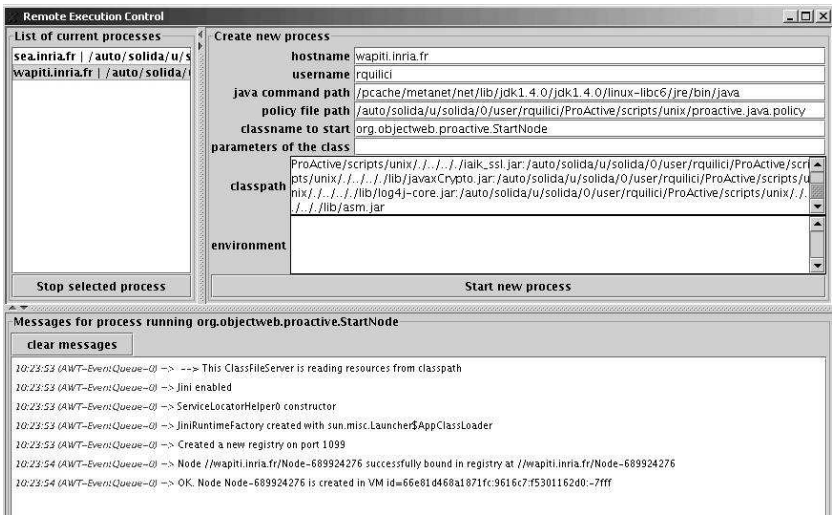


Fig. 9. Interactive creation of a new JVM and associated node

their inclusion. Of course, the *IC2D* monitor itself is still useful so as to visualize the underlying infrastructure and all activities. At this point, we will need to extend the *IC2D* monitor, so as to provide a way to graphically show all activities that are part of the same component: in this way, it will be possible to trigger the migration of a component as a whole, by using the usual drag-and-drop facility.

4.4 Security Enforcement

We now describe the fourth step mentioned in section 4.1. Grid applications must be able to cope with large variations in deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. In the same way as the deployment is not tied up in the source code, we provide a similar solution regarding the security, so as it becomes easily configurable in a flexible, and abstract manner. Overall, we propose a framework allowing:

- declarative security attributes (Authentication, Integrity, Confidentiality), outside any source code and away from any API;
- policies defined hierarchically at the level of administrative domain,
- dynamic security policy, taking into account the nature (private or public) of the underlying networks;
- dynamically negotiated policies (for multi-principals applications),
- policies for remote creation and migration of activities.

A Hierarchical Approach to Grid Security A first decisive feature allows to define application-level security on Virtual Nodes, those application-level deployment abstractions:

Definition 1. *Virtual Node Security*
Security policies can be defined at the level of Virtual Nodes. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to JVMs, and Hosts.

As such, virtual nodes are the support for intrinsic application level security. If, at design time, it appears that a process always requires a specific level of security (e.g. authenticated and encrypted communications at all time), then that process should be attached to a virtual node on which those security features are imposed. It is the designer responsibility to structure his/her application or components into virtual node abstractions compatible with the required security. Whatever deployment occurs, those security features will be maintained.

The second decisive feature deals with a major Grid aspect: deployment-specific security. The issue is actually twofold:

1. allowing organizations (security domains) to specify general security policies,
2. allowing application security to be specifically adapted to a given deployment environment.

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure; they are organized in a hierarchical manner. They are the logical concept allowing to express security policies in a hierarchical way.

Definition 2. *Declarative Domain Security*
Fine grain and declarative security policies can be defined at the level of Domains. *A Security Domain is a domain to which a certificate and a set of rules are associated.*

This principle allows to deal with the two issues mentioned above:
(1) the administrator of a domain can define specific policy rules that must be obeyed by the applications running within the domain. However, a general rule expressed inside a domain may prevent the deployment of a specific application. To solve this issue, a policy rule can allow a well-defined entity to weaken it. As we are in a hierarchical organization, allowing an entity to weaken a rule means allowing all entities included to weaken the rule. The entity can be identified by its certificate;
(2) a Grid user can, at the time he runs an application, specify additional security based on the domains being deployed onto, directly in his deployment descriptor for those domains.

Finally, as active objects are active and mobile entities, there is a need to specify security at the level of such entities.

Definition 3. *Active Object Security*
Security policies can be defined at the level of Active Object. Upon migration of an activity, the security policy attached to that object follows.

In open applications, e.g. several principals interacting in a collaborative Grid application, a JVM (a process) launched by a given principal can actually host an activity executing under another principal. The principle above allows to keep specific security privileges in such case. Moreover, it can also serve as a basis to offer, in a secure manner, hosting environments for mobile agents.

Interactions Definition Security policies are able to control all the *interactions* that can occur when deploying and executing a multi-principals Grid application. With this goal in mind, interactions span over the creation of processes (JVM in our case), to the monitoring of activities (ActiveObjects) within processes, including of course the communications. Here is a brief description of those interactions:

- JVMCreation (JVMC): creation of a new JVM process
- NodeCreation (NC): creation of a new Node within a JVM (as the result of Virtual Node mapping)
- CodeLoading (CL): loading of bytecode within a JVM
- ActiveObjectCreation (AOC): creation of a new activity (active object) within a Node
- ActiveObjectMigration (AOM): migration of an existing activity object to a Node
- Request (Q), Reply (P): communications, method calls and replies to method calls
- Listing (L): list the content of an entity; for Domain/Node provides the list of Node/Active Objects, for Active Object allows to monitor its activity.

One must be able to express policies in a rather declarative manner. The general syntax to provide security rules, to be placed within security policy files attached to applications (for instance, see the `security` tag within the deployment descriptor), is the following:

```
Entity[Subject] -> Entity [Subject]
                                : Interaction # [SecurityAttributes]
```

Being in a PKI infrastructure, the subject is a certificate, or credential. Other "elements" (Domain, Virtual Node, Object) are rather specific to Grid applications and, in some cases, to the object-oriented framework. An "entity" is an element on which one can define a security policy. "Interaction" is a list of actions that will be impacted by the rule. Finally, security attributes specify how, if authorized, those interactions have to be achieved.

In order to provide a flavor of the system, we consider the following example.

```
Domain[inria.fr] -> Domain[ll.cnrs.fr] : Q,P # [+A,+I,?C]
```

The rule specifies that between the domain *inria.fr* (identified by a specific certificate) and the parallel machine *ll.cnrs.fr*, all communications (reQuests, and rePlies) are authorized, they are done with *authentication* and *integrity*, *confidentiality* being accepted but not required.

Security Negotiation As a Grid operates in decentralized mode, without a central administrator controlling the correctness of all security policies, these policies must be *combined*, *checked*, and *negotiated* dynamically.

During execution, each activity (Active Object) is always included in a *Node* (due to the Virtual Node mapping) and at least in one *Domain*, the one used to launch a JVM (D_0). Figure 10 hierarchically represents the security rules that

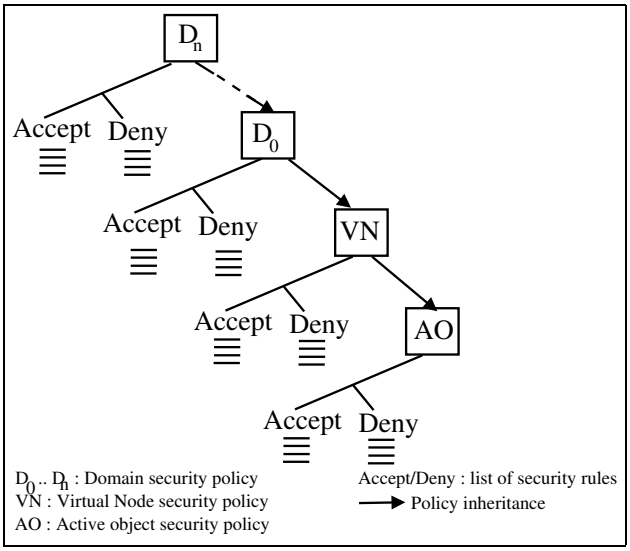


Fig. 10. Hierarchical security levels

can be activated at execution: from the top, hierarchical domains (D_n to D_0), the virtual node policy (VN), and the Active Object (AO) policy. Of course, such policies can be inconsistent, and there must be clear principles to combine the various sets of rules.

There are three main principles: (1) choosing the *most specific rules* within a given domain (as a single Grid actor is responsible for it), (2) an interaction is valid only if all levels accept it (absence of weakening of authorizations), (3) the security attributes retained are the most constrained based on a partial order (absence of weakening of security). Before starting an interaction, a *negotiation* occurs between the two entities involved.

In large scale Grid applications, migration of activities is an important issue. The migration of Active Objects must not weaken the security policy being applied. When an active object migrates to a new location, three cases may happen :

- the object migrates to a node belonging to the same virtual node and included inside the same domain. In this case, all already negotiated sessions remain valid.
- the object migrates to a known node (created during the deployment step) but which belongs to another virtual node. In this case, all already negotiated sessions can be invalid. This kind of migration imposes re-establishing the object policy, and upon a change, re-negotiating with interacting entities.
- The object migrates to an unknown node (not known at the deployment step). In this case, the object migrates with a copy of the application security policy. When a secured interaction will take place, the security system

retrieves not only the object's application policy but also policies rules attached to the node on which the object is to compute the policy.

5 Conclusion and Perspectives

In summary, the essence of our proposition, presented in this paper, is as follows: a distributed object oriented programming model, smoothly extended to get a component based programming model (in the form of a 100% Java library); moreover this model is "grid-aware" in the sense that it incorporates from the very beginning adequate mechanisms in order to further help in the deployment and runtime phases on all possible kind of infrastructures, notably secure grid systems. This programming framework is intended to be used for large scale grid applications. For instance, we have succeeded to apply it for a numerical simulation of electromagnetic waves propagation, a non embarrassingly parallel application [21], featuring visualization and monitoring capabilities for the user. To date, this simulation has successfully been deployed on various infrastructures, ranging from interconnected clusters, to an intranet grid composed of approximatively 300 desktop machines. Performances compete with a previous existing version of the application, written in Fortran MPI. The proposed object-oriented approach is more generic and features reusability (the component-oriented version is under development, which may further add dynamicity to the application), and the deployment is very flexible.

We are conducting further works in several but complementary directions that are needed in grid computing, mainly:

- checkpointing and message logging techniques are in the way of being incorporated into the *ProActive* library. Indeed, we will as such be able to react to versatility of machines and network connections, without having to restart all the application components. Several similar works are under way ([22] for instance). The original difficulty we are faced with, is that it is possible to checkpoint the state of an active object only at specific points: only between the service of two requests (for the same reason which explains why the migration of active objects is weak). Nevertheless, we provide an hybrid protocol combining communication induced checkpointing and message logging techniques, which is adapted to the non-preemptibility of processes. This protocol ensures strong consistency of recovery lines, and enables a fully asynchronous recovery of the distributed system after a failure.
- *ProActive* components that wrap legacy codes, and in particular, parallel (MPI) native codes, are being defined and implemented (see [23] for related approaches). Of course, the design aims at enabling such components to interact with 100% *ProActive* components.

Overall, we target numerical code coupling, combination of numerical simulations and visualization, collaborative environments in dedicated application domains (see [24]), etc. The aim is to use our grid component model as a software bus for interactions between some or all of the grid components.

Indeed, the presented approach does not specifically target high-level tools appropriate for scientists or engineers who may not have a computer science background. In this respect, our objective is to succeed to incorporate the *IC2D* tools suite into grid portals, such as the Alliance portal [25]. An other complementary on going work for this class of users is to enable *ProActive* components to be published as web services (and enable those web service enabled components interact using SOAP). Notice that it does not prevent a service to be implemented as one or several hierarchical ProActive components, i.e. as the result of a recursive composition of 100% *ProActive* components, internally interacting only through *ProActive*. Then, within such portals, end-users could rely on workflow languages such as WSFL or BPEL4WS to compose applications by simply integrating some of the published components at a *coarse-grain level*.

In this way, those coarse-grain web service enabled components could provide the usual service-oriented view most users are familiar with. But as those instanciated components may use stateful resources, encompass possibly complex compositions of activities and data, we claim that the object and component oriented programming model we propose is adequate to 'internally' program and deploy those hierarchical components.

References

1. Baude, F., Caromel, D., Huet, F., Mestre, L., Vayssière, J.: Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In: 11th IEEE International Symposium on High Performance Distributed Computing. (2002) 93–102
2. Attali, I., Caromel, D., Contes, A.: Hierarchical and declarative security for grid applications. In: 10th International Conference On High Performance Computing, HIPC. Volume 2913., LNCS (2003) 363–372
3. Baduel, L., Baude, F., Caromel, D.: Efficient, flexible, and typed group communications in java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, ACM Press (2002) 28–36
4. Baude, F., Caromel, D., Morel, M.: From distributed objects to hierarchical grid components. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003. Volume 2888., LNCS (2003) 1226–1242
5. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02). (2002)
6. Bruneton, E., Coupaye, T., Stefani, J.B.: Fractal web site. `http://fractal.objectweb.org` (2003)
7. Gannon, D., Bramley, R., Fox, G., Smallen, S., i, A.R., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., indaraju, M.G., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N.: Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. Cluster Computing **5** (2002)

8. Fox, G., Pierce, M., Gannon, D., Thomas, M.: Overview of Grid Computing Environments. Global Grid Forum document, http://forge.gridforum.org/projects/ggf-editor/document/GFD-I.9/en/1 (2003)
9. Grimshaw, A., Wulf, W.: The Legion Vision of a World-wide Virtual Computer. Communications of the ACM **40** (1997)
10. Humphrey, M.: From Legion to Legion-G to OGSI.NET: Object-based Computing for Grids. In: NSF Next Generation Software Workshop at the 17th International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, IEEE Computer Society (2003)
11. Bramley, R., Chin, K., Gannon, D., Govindaraju, M., Mukhi, N., Temko, B., Yochuri, M.: A Component-Based Services Architecture for Building Distributed Applications. In: 9th IEEE International Symposium on High Performance Distributed Computing. (2000)
12. Denis, A., Pérez, C., Priol, T.: Achieving portable and efficient parallel corba objects. Concurrency and Computation: Practice and Experience **15** (2003) 891–909
13. Denis, A., Prez, C., Priol, T., Ribes, A.: Padico: A component-based software infrastructure for grid computing. In: 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS2003). (2003)
14. Caromel, D., Belloncle, F., Roudier, Y.: The C++// Language. In: Parallel Programming using C++. MIT Press (1996) 257–296 ISBN 0-262-73118-5.
15. Baude, F., Caromel, D., Sagnol, D.: Distributed objects for parallel numerical applications. Mathematical Modelling and Numerical Analysis Modelisation, special issue on Programming tools for Numerical Analysis, EDP Sciences, SMAI **36** (2002) 837–861
16. Bull, J., Smith, L., Pottage, L., Freeman, R.: Benchmarking Java against C and Fortran for Scientific Applications. In: Joint ACM Java Grande - ISCOPE 2001 Conference, Palo Alto, CA, ACM Press (2001)
17. Caromel, D.: Towards a Method of Object-Oriented Concurrent Programming. Communications of the ACM **36** (1993) 90–102
18. Maisonneuve, J., Shapiro, M., Collet, P.: Implementing references as chains of links. In: 3d Int. Workshop on Object-Orientation in Operating Systems. (1992)
19. Dincer, K.: Ubiquitous message passing interface implementation in Java: JMPI. In: Proc. 13th Int. Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing, IEEE (1999)
20. OASIS: ProActive web site, http://www.inria.fr/oasis/ProActive/. (2004)
21. Baduel, L., Baude, F., Caromel, D., Delbe, C., Gama, N., Kasmi, S.E., Lanteri, S.: A parallel object-oriented application for 3d electromagnetism. In: IEEE International Symposium on Parallel and Distributed Computing, IPDPS. (2004)
22. Bouteiller, A., Cappello, F., Herault, T., G.Krawezik, Marinier, P.L., Magniette, F.: A fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In: ACM/IEEE International Conference on Supercomputing SC 2003. (2003)
23. Li, M., Rana, O., Walker, D.: Wrapping MPI-based Legacy Codes as Java/CORBA components. Future Generation Computer Systems **18** (2001) 213–223
24. Shields, M., Rana, O., Walker, D.: A Collaborative Code Development Environment for Computational Electro-Magnetics. In: IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software, Kluwer Academic Publishers (2001) 119–141
25. : The Alliance Portal. http://www.extreme.indiana.edu/xportlets/project/index.shtml (2004)