From Distributed Objects to Hierarchical Grid Components

Françoise Baude, Denis Caromel, and Matthieu Morel

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia-Antipolis
2004, Route des Lucioles, BP 93
F-06902 Sophia-Antipolis Cedex - France
{Francoise.Baude,Denis.Caromel,Matthieu.Morel}@sophia.inria.fr

Abstract. We propose a parallel and distributed component framework for building Grid applications, adapted to the hierarchical, highly distributed, highly heterogeneous nature of Grids. This framework is based on ProActive, a middleware (programming model and environment) for object oriented parallel, mobile, and distributed computing. We have extended ProActive by implementing a hierarchical and dynamic component model, named Fractal, so as to master the complexity of composition, deployment, re-usability, and efficiency of grid applications. This defines a concept of Grid components, that can be parallel, made of several activities, and distributed. These components communicate using typed one-to-one or collective invocations.

Keywords: Active objects, components, hierarchical components, grid computing, deployment, dynamic configuration, group communications, ADL.

1 Introduction

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

For Grid applications development, there is indeed a need also to smoothly, seamlessly and dynamically integrate and deploy autonomous software, and for this provide a *glue* in the form of a software bus. In this sense, we essentially address the second group of Grid programmers such as defined in [1]: first group of users are end users who program pre-packaged Grid applications by using a simple graphical or Web interface; the second group of grid programmers are those that know how to build Grid applications by composing them from existing application "components"; the third group consists of the researchers that build the individual components.

R. Meersman et al. (Eds.): CoopIS/DOA/ODBASE 2003, LNCS 2888, pp. 1226–1242, 2003.

We share the goal of providing a component-based high-performance computing solution with several projects such as: CCA [1] with the CCAT/XCAT toolkit [2] and Ccaffeine framework, Parallel CORBA objects [3] and GridCCM [4]. But, to our knowledge, our contribution is the first framework featuring *hierarchical* distributed components. This clearly helps in mastering the complexity of composition, deployment, re-usability required when programming and running large-scale distributed applications.

We propose a parallel and distributed component framework for building meta-computing applications, that we think is well adapted to the hierarchical, highly distributed, highly heterogeneous nature of grid-computing. This framework is based on ProActive, a Java-based middleware (programming model and environment) for object oriented parallel, mobile and distributed computing. ProActive has proven to be relevant for grid computing [5] especially due to its deployment and monitoring aspects [6] and its efficient and typed collective communications [7]. We have succeeded in defining a component model for ProActive, with the implementation of the Fractal component model [8,9], mainly taking advantage of its hierarchical approach to component programming.

Fractal is a general software composition model, implemented as a framework that supports component-based programming, including hierarchical components (type) definition, configuration, composition and administration. Fractal is an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. Indeed, a system administrator, a system integrator or an application developer may need to dynamically construct a system or service out of existing components, whether in response to failures, as part of the continuous evolution of a running system, or just to introduce new applications in a running system (a direct generalization of the dynamic binding used in standard distributed client-server applications). Nevertheless, the requirements raised by distributed environments are not specifically addressed by the Fractal model. Not because this is not an issue, but, because, according to the Fractal specification, a primitive or hierarchical fractal component *may be* a parallel and distributed software. So, our work also yields to a new implementation of the Fractal model that explicitly provides parallel and distributed Fractal components.

The main achievement of this work is to design and implement a concept of *Grid Components*. Grid components are recursively formed of either sequential, parallel and/or distributed sub-components, that may wrap legacy code if needed, that may be deployed but further reconfigured and moved – for example to tackle fault-tolerance, load-balancing, adaptability to changing environmental conditions.

Below is a typical scenario illustrating the usefulness of our work. Assume a complex grid software be formed of several services, say of other software (a parallel and distributed solver, a graphical 3D renderer, etc). The design of such a software is very much 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 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 on-going computations may just be moved (the solver for instance), alas 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. And once the new graphical renderer is launched, we re-bind the software, so as it now uses this new instance.

This paper is organized as follows: after an introduction on parallel and distributed programming with ProActive, and on the Fractal component model, the principles and design of the proposed parallel and distributed component model are presented. The implementation and an example are described in section 4, while section 5 makes a comparison with related work before concluding.

2 Context

2.1 Distribution, Parallelism, Mobility, and Deployment with ProActive

The ProActive middleware is a 100% Java library (LGPL) [10] aiming to achieve seamless programming for concurrent, parallel, distributed and mobile computing. The main features regarding the programming model are:

- a uniform *active object* programming pattern
- remotely accessible objects, via method invocation
- asynchronous communications with automatic synchronization (automatic futures for results of remote method calls). Note that asynchronicity enables to use one-way calls for transmitting events.
- group communications, which enable to trigger method calls on a distributed group of active objects of the same compatible type, with a dynamic generation of groups of results. It has been shown in [7] that this group communication mechanism, plus a few synchronization operations (`WaitAll, WaitOne`, etc), provides quite similar patterns for collective operations such as those available in e.g. MPI, but in a language centric approach. Here is an example:

```
//Object 'a' of class A is an active remote object
V v = a.foo(param);
// remotely calls foo on object a

v.bar();
// automatically blocks on v.bar()
// until the result in v gets available.
```

```
// ag is a group of active objects,
// of types compatible with A
V v = ag.foo(param);
// calls foo on each group member
// with some optimisation at serialization time
// V is automatically built as a group of results
v.bar();
// executes as soon as individual results
// of foo calls return
```

– migration (mobile computations): An active object with its pending requests (method calls), its futures, its passive (mandatory non-shared) objects may migrate from JVMs to JVMs. The migration may be initiated from outside through any public method but it is the responsibility of the active object to execute the migration (weak migration). Automatic, transparent (and optimized) forwarding of requests and replies provide location transparency, as remote references towards active mobile objects remain valid.

We are faced with the common difficulties in deployment regarding launching a ProActive application in its environment. We succeed in completely avoid scripting for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code such as to gain in flexibility [6] as follows (see figure 8 for an example):

– XML Deployment Descriptors. Active objects are remotely created on JVMs, but *virtual nodes* are manipulated inside the program, instead of URLs of JVMs. The *mapping* of virtual nodes to effective JVMs is managed externally through those descriptors. Descriptors also permit to define how to launch JVMs.
– Interfaces with various protocols: `rsh`, `ssh`, `LSF`, `Globus`, `Jini`, `RMIregistry` enable to effectively launch, register or discover JVMs according to the needs specified in the descriptor.
– Graphical visualization and monitoring of any ongoing ProActive application is possible through a ProActive application called *IC2D* (Interactive Control and Debugging of Distribution). In particular, IC2D enables to migrate executing tasks by a graphical drag-and-drop, and to create additional JVMs.

2.2 The Fractal Component Model

The Fractal component model provides an homogeneous vision of software systems structure with a few but well defined concepts such as component, controller, content, interface, binding. It also exhibits distinguishing features that have proven useful for the present work: it is recursive – components structure is auto-similar at any arbitrary level (hence the name 'Fractal'); it is completely reflexive, i.e., it provides introspection and inter-cession capabilities on components structure. These features allow for a uniform management of both the

so-called business and technical components (which is not the case in industrial component frameworks such as EJB [11] or CCM [12] which only deal with business components).

A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, which are under the control of the controller of the enclosing component. This allows for hierarchic components, in the sense that components may be nested at any arbitrary level. Fractal distinguishes *primitive* components (typically associated to a Java class implementing functional services) and *composite* components that only serve to build hierarchies of components, but without implementing themselves functional services.

A component can interact with its environment through *operations* at identified access points, called *interfaces*. As usual, interfaces can be of two sorts: *client* and *server*. A server interface can receive operation invocations (and return operation results of two-way operations), while a client interface emits operations. A *binding* is a connection between components, and more precisely between a client and a server interface. The Fractal model comprises bindings for composite and primitive components. Bindings on client ports of primitive components are typically implemented as language-level bindings (e.g. through type compatible variable affectations of interface references). The type of a binding might be a *collective* one or a *single* one (as default). In case of a collective one, a component may need, for achieving its functional work, to use (thus be bound to) a collection of components, instead of to a single component, all offering the needed interface.

A component controller embodies the control behavior associated with a particular component. Of importance is the following control: suspend (stop) and resume activities of the components in its content. Stopping then resuming is mandatory in order to dynamically change the binding between components or the inclusion of components. The important fact is that all such non-functional calls (stopping, resuming, binding, etc) propagates recursively to each internal component. This prevents the user manually triggering the same call on each sub-sub-...-sub component.

3 From Active Objects to Parallel, Distributed, and Hierarchical Components

3.1 Evaluation of the Needs

A component must be aware of parallelism and distribution as we aim at building a grid-enabled application by hierarchical composition; indeed, we need a glue to couple codes that probably are parallel and distributed codes as they require high performance computing resources. Thus 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 1 summarizes the three different cases for the structure of a Grid component. For a composite built up as a collection of components providing common services, (fig. 1 c)) *collective communications* are essential, for ease of programming and efficiency purposes.

As general requirements, because we target high performance grid computing, it is very important to efficiently implement point-to-point and group method invocations, manage the deployment complexity of those components distributed all over the grid and possibly debug, monitor and reconfigure those running components – across the world.

3.2 ProActive Components

In the sequel, we describe the component framework we have designed and implemented using both Fractal and ProActive. It enables to couple parallel and distributed codes directly programmed using the Java ProActive library. A synthetic definition of what is a ProActive component is given below.

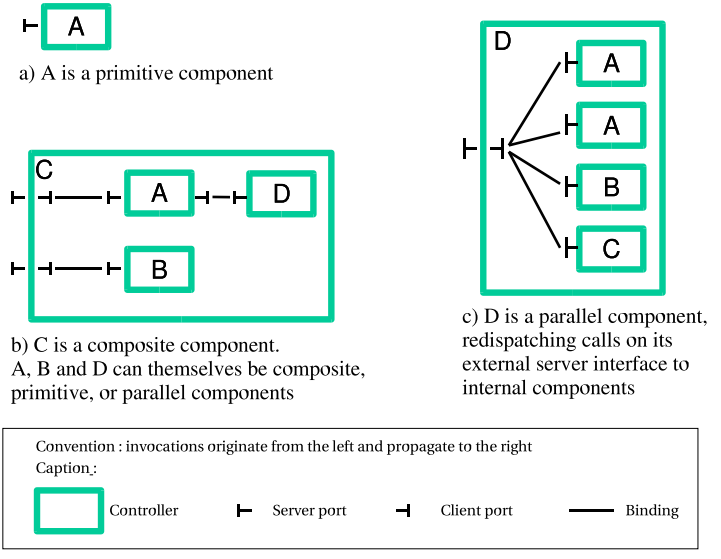


Fig. 1. The various basic architectures for a Grid component

Definition of a ProActive component:

- It is formed from one (or several) Active Objects, executing on one (or several) JVM
- 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 redispatching calls to its external server interfaces towards its inner components.
- It communicates with other components through 1-to-1 or group communications.

ProActive components 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

Deployment of ProActive Components. Components are a way to globally manipulate distributed and running activities, and in this context, obviously, the concept of *virtual node* is a very important abstraction. The additional need regarding the ones already solved by the deployment of active objects, is to be able to *compose virtual nodes*: a composite component is defined through a number of sub-components that already define their proper usage and mapping of virtual nodes. What should the mapping of the composite be ? For instance on fig. 2, when grouping two components in a new composite one, assume that each of the two sub-components, named respectively A and B, requires to be deployed respectively on VNa (further associated to 3 JVMs through the deployment descriptor) and the same for VNb (3 other JVMs). The question is how to define the mapping of the new composite ? Either distributed mapping is required (see fig. 2 a)) meaning that VNa and VNb must respectively launch different JVMs (a total of 6); or, a co-allocated mapping (see fig. 2 b)) where we try to co-locate as much as possible one activity acting on behalf of sub-component A and one activity acting on behalf of sub-component B within the composite C (on the example, only 3 JVMs need to be used).

Composition of virtual nodes is thus a mean to control the distribution of composite components.

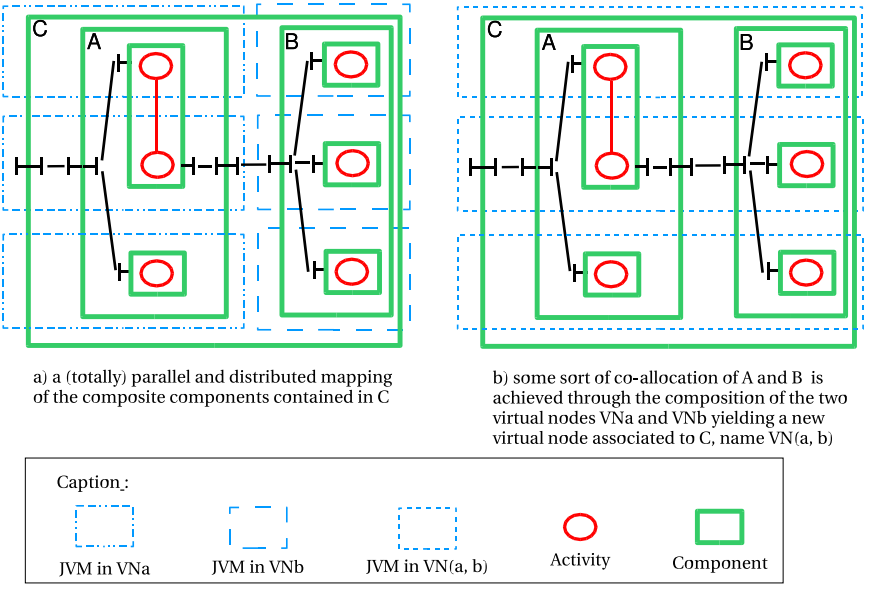


Fig. 2. Components versus Activities and JVMs

4 Implementation and Example

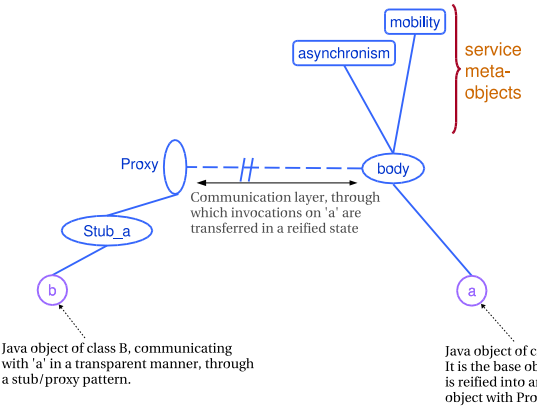
Fractal, along with the specification of a component model, also defines an API in Java. There is a reference implementation, called Julia, and we propose a new implementation, based on ProActive (thus providing all services offered by the Fractal library).

4.1 Meta-object Protocol

ProActive is based on a Meta-Object Protocol (MOP)(Figure 3), that allows to add many aspects on top of standard Java objects, such as asynchronism and mobility. Active objects are referenced through stubs, and the communication with them is done in the same manner, would they actually be remote or local.

The same idea is used to manage components: we just add a set of meta-objects in charge of the component aspect (Figure 4). Of course, the standard ProActive stub (that gives a representation of type A on the figure) is not used here, as we manipulate components. In Fractal, a reference on a component is of type `ComponentIdentity`, so we provide a new stub (that we call representative), of type `ComponentIdentity`, that references the actual component. All standard Fractal operations can then be performed on the component.

In our implementation, because we make use of the MOP's facilities, all components are constituted of one active object (at least), are they composite or primitive. Of course, if the component is a composite, and if it contains other



Java objects 'b' and 'a' can be in different virtual machines (the network being represented here between the proxy and the body, though the call might be local).Object 'b' has a reference on active object 'a' (of type `A`) through a stub (of type `A`) and a proxy. When 'b' invokes a method on 'stub_a',the call is forwarded though the communication layer (possibly through a network) to the body of the active object. At this point, the call can be intercepted by the meta-objects, possibly leading to some induced actions, and then the call is forwarded to the base object 'a'.

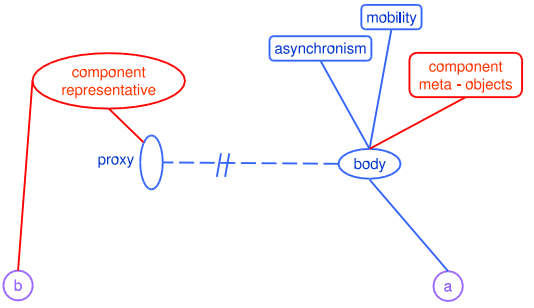
Fig. 3. ProActive's Meta-Object Protocol.

components, then we can say it is constituted of several active objects. Also, if the component is primitive, but the programmer of this component has put some code within it for creating new active objects, the component is again constituted of several active objects.

4.2 Integration within ProActive

To integrate the component management operations into the ProActive library, we just make use of the extensible architecture of the library. This way, components stay fully compatible with standard active objects and as such, inherit from the features active objects have: mobility, security, deployment, etc.

A particular point for the integration of Fractal and ProActive to succeed is the management of component requests besides functional requests. Reified method calls, when they arrive in the body, are directed towards the queue of requests. We assume FIFO is the processing policy. The processing of the requests in the queue is dependent on the nature of this request, and corresponds to the following algorithm :



A new set of meta-objects, managing the component aspect (constituting the controller of the component, in the Fractal terminology), is added to the active object 'a', and 'b' can manipulate the component based on 'a' through a specific stub, called *component representative*, of type `ComponentIdentity`. If the component is primitive, 'a' contains the functional code of the component.

Fig. 4. Component meta-objects and component representative.

```
loop
  if componentLifeCycle.isStarted()
      get next request
      // all requests are served
  else if componentLifeCycle.isStopped()
      get next component controller request
      // only component requests are served
  ;
  if gotten request is a comp. life cycle request
      if startFc --> set started = true ;
      if stopFc --> set started = false ;
  ;
;
```

Note that, in the stopped state, only controller requests are served. This means that a standard ProActive call, originating from a standard ProActive stub, will not be processed in the "stopped" state (but it will stay in the queue).

4.3 Collective Ports, Group Communications, and Parallel Components

The implementation of collective ports is based on the ProActive groups API (cf. [7]). According to the Fractal specification, this type of interfaces only has

sense on client interface, that would like to be bound to several server interfaces. Besides, one server interface can always be accessed by several client interfaces, the calls being processed sequentially. Specifying a server interface as "collective" wouldn't change its behavior.

The ProActive groups API allowing group communication in a transparent manner, the implementation of the collective interfaces slightly differs from the Fractal specification: instead of creating one new interface with an extended name for each member of the collection, we just use one interface (that is actually a group). Collective bindings are then performed transparently as if they were multiple sequential bindings on the same interface. Using a collective server interface will then imply using the ProActive group API formalism, including the possibility to choose between *scattering* and *broadcasting* of the calls [7]. A feature is that unbinding operations on a collective interface will result in the removal of all the bindings of the collection.

Furthermore, because we target largely distributed and parallel applications, we introduce a new type of component : *parallel components* (Figure 1 c)). These components are composite components, as they encapsulate other components. Their specificity relies in the behavior of their external server interfaces. These interfaces are connectable through a group proxy to the internal components' interfaces of the same type. This means that a call to the parallel component will be dispatched and forwarded to a set of internal components, that will process the requests in a parallel manner (see figure 5 a)).

4.4 Example

We present hereby an example of a component system built using our component model implementation.

Consider the following music diffusion system : a cd-player reads music files from a cd, and transmits them to a set of speakers situated in different rooms. Those speakers can convert music files into music we can listen to. They are incorporated in a parallel component, thus providing a single access interface to them (instead of connecting the cd player's output to each of the speakers).

Figure 6 gives an overview of the system, and represents the component model.

The system can be configured using the ADL (Architecture Description Language) that we provide for the components (Figure 7, coupled with the deployment descriptor, describing the physical infrastructure (Figure 8)).

When using the ADL, the configuration of the components is read from the descriptors, and the components are automatically instantiated, assembled and bound. Figure 9 shows an example of code used to manipulate the components, including instantiation, control and functional operations.

5 Related Work

We compare with closest related work in spirit, i.e. high-performance computing with composition of software components.

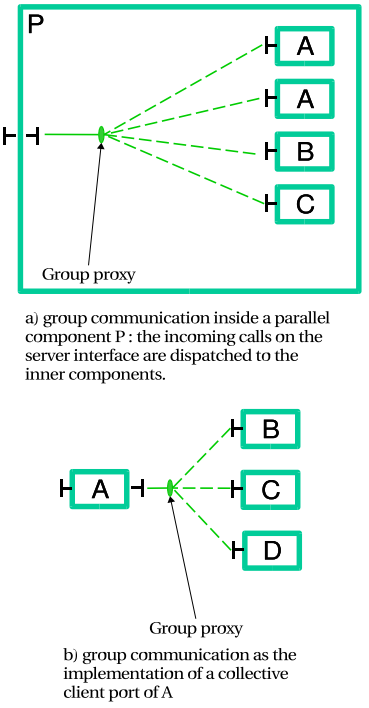


Fig. 5. Group communications allowing collective bindings and parallel components

CCA. The Common Component Architecture [1] is an initiative to define minimal specification for scientific components, targeting parallel and distributed infrastructures. Ideas are drawn from CCM for the sake of defining components by provide/use ports, calls/events through the usage of a Scientific IDL (SIDL). A toolkit of the CCA specification, called CCAT [2], provides a framework for applications defined by binding CCA-enabled components, in which all services (directory, registry, creation, connection, event) are themselves CCA components (wrapping external services). An instance of this framework, XCAT, permits to describe a component and its deployment using an XML document, which looks very similar to what we have also defined and implemented for ProActive components. In this XML-oriented implementation of CCA, the communication protocol used to implement the remote procedure call between a uses port method invocation and the connected provides port remote objects is based on SOAP. The main drawback of CCA is that the composition model is not related to any specific underlying distributed object oriented model so that the user lacks a clear and precise model of the composition (which is as important as having a clear and precise programming model).

Corba Parallel Objects. The Parallel Corba model [3] targets the coupling of objects whose execution model is parallel (in practice, a parallel object is incar-

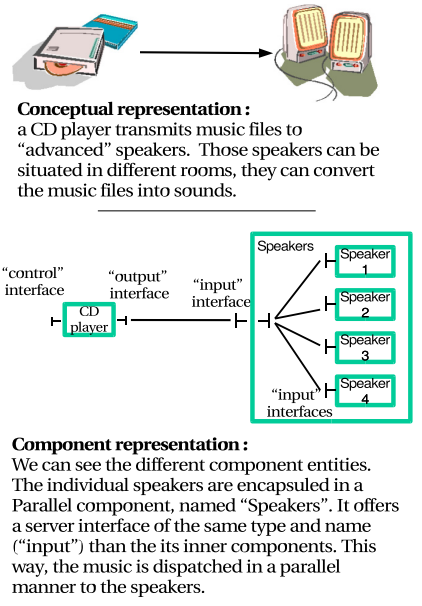


Fig. 6. A music diffusion system based on components

nated by a collection of sequential objects, and the execution model is SPMD; thus invoking a method on a parallel object invokes the corresponding method on all the objects of the collection, by scattering and redistributing arguments if needed). An implementation, PaCO++, achieves portability through the usage of standard CORBA IDL for object interactions. Notice that parallel and distribution issues are separated, as CORBA is only used to couple distributed codes, and parallel computations are usually managed with MPI. This is obviously an obstacle to easy grid computing.

GridCCM. GridCCM [4], a Parallel Corba component model, is a natural extension of PaCO++ motivated by the fact that a code coupling application can be seen as an assembly of components; however, most software component models (except PaCO++) only support sequential component. In order to have transparency in the assembly of components, a design choice was to make effective communications between parallel components be hidden to the application designer, by introducing collective ports that look like to be ordinary single-point ports. We propose the same sort of facility: ProActive components may also be built as parallel components by providing and using collective interfaces.

None of those approaches define hierarchical components as we have presented here. Moreover, we can encompass both parallel components in an SPMD style, or more generally parallel and distributed components following an MIMD execution model. We emphasize that we provide a unique infrastructure for functional and parallel calls and for component management, which is an alternative

```
COMPONENTS DESCRIPTOR
  Primitive–component  "cd–player"
      implementation = "CdPlayer"
      // name of the Java class with the functional code of the cd player
      VN = Node–player  //see deployment descriptor
      provides
          interface "control"
              signature = soundssystem.PlayerFacade
      requires
          interface "output"
              signature = soundssystem.Output
  Parallel–component  "speakers"
      VN = Node–speakers
      // the parallel component is just a facade to the real speakers
      provides
          interface "input"
              signature = soundssystem.Input
      contains
          primitive–component  "speaker"
              implementation = "Speaker"
              // functional code of the speaker
              VN = Node–speaker  (cyclic)  // see deployment descriptor
              // deployment descriptor will specify the location of
              // the instances (thus their number)
              provides
                  interface "input"
                      signature = soundssystem.Input
          Bindings
          // bindings to inner components are automatic for parallel
          // components between server interfaces of the same name
  Bindings
  // between client and server interfaces of the components
    bind "cd–player.output" to "speakers.input"
```

Fig. 7. Using the ADL to describe a component system (format is converted from XML)

to what is for instance done in GridCCM [4] (MPI, openMP, etc.,) for functional and parallel codes, and Corba for component management – binding, deployment, life-cycle management, ...).

6 Conclusion and Perspectives

We have successfully defined and implemented a component framework for ProActive, by applying the Fractal component model, mainly taking advantage of its hierarchical approach to component programming.

This defines a concept of what we have called *Grid components*. Grid components are formed of parallel and distributed active objects, features mobility, typed one-to-one or collective service invocations and a flexible deployment model. They also features flexibility and dynamicity at the component definition level.

We are working on the design of generic wrappers written in ProActive whose aim is to encaspulate legacy parallel code (usually Fortran-MPI or C-MPI codes).

We are also working on GUI-based tools to help the end-user to manipulate grid component based applications. Those tools will extend the IC2D monitor, which already helps in dynamically changing the deployment defined by deployment descriptors (cf. figure 8): acquire new JVMs, drag-and-drop active objects on the grid. We will provide *interactive* dynamic manipulation and monitoring

```
DEPLOYMENT DESCRIPTOR
VirtualNodes   // names of the virtual nodes
    VirtualNode name = "Node−player"
    VirtualNode name = "Node−speakers"
    VirtualNode name = "Node−speaker" − cyclic
    // cyclic: i.e. there will actually be several JVMs
Deployment  // what is behind the names of the virtual nodes
    mapping
    // correspondance between the names of the VNs and the JVMs
        Node−player −−> JVM1
        Node−speakers −−> JVM1
        Node−speaker −−> {JVM2, JVM3, JVM4}
        // 1 VN can be mapped onto a set of JVMs
    JVMs
        JVM1 created by process "linuxJVM"
        JVM2 created by process "rsh−computer1"
        JVM3 created by process "rsh−computer2"
        JVM4 created by process "globus−computer1"
Infrastructure
// how and where the JVMs specified above are created
    process−definition "linuxJVM"
    // this process creates a JVM on the current host
        JVMProcess class=JVMNodeProcess
    process−definition "rsh−computer1"
    // this process establishes an rsh connection
    // and starts a JVM on the remote host
    // (using the previously defined process "linuxJVM"
        rshProcess class=RSHProcess host="computer1"
        // computer1 could be in room1
        processReference = "linuxJVM"
    process−definition "rsh−computer2"
        rshProcess class=RSHProcess host="computer2"
        // computer2 could be in room2
        processReference = "linuxJVM"
    process−definition "globus−computer1"
        globusProcess class=GlobusGramProcess host="globus1"
        // globus1 could be in a room abroad
        processReference = "linuxJVM"
```

Fig. 8. Using the deployment descriptor to describe the physical infrastructure of a component system (format is converted from XML)

```
// CREATE THE COMPONENTS (for example speakers and cd_player)
ComponentIdentity speakers =
ProActive.newActiveComponent(speakers_parameters);
ComponentIdentity cd_player =
ProActive.newActiveComponent(cd_player_parameters);
// If the ADL is used, components instances can be retreived
// through the ComponentsLoader class :
// ComponentIdentity speakers =
// ComponentsLoader.getComponent("speakers");

// BIND THE COMPONENTS (Using the BindingController)
// (this is automatically done when using the ADL)
((BindingController)cd_player
.getFcInterface(BindingController.BINDING_CONTROLLER))
.bindFc(« output », speakers.getFcInterface(« input »);

// START THE LIFE CYCLE OF THE COMPONENTS
// (ENABLE THE COMPONENTS), using the LifeCycleController
((LifeCycleController)speakers
.getFcInterface(LifeCycleController.LIFECYCLE_CONTROLLER))
.startFc();
// this call is recursive, as the component contains other
// components (it also starts the inner components)

((LifeCycleController)cd_player
.getFcInterface(LifeCycleController.LIFECYCLE_CONTROLLER))
.startFc();


// INVOKE SOME ACTIONS ON FUNCTIONAL INTERFACES
// invoking a method of the Input interface
((Input)speakers
.getFcInterface(«input»))
.newMusic (music.mp3);

// invoking a method of the PlayerFacade interface
((PlayerFacade)cd_player
.getFcInterface(« control »))
.play();
```

Fig. 9. Using the API to manipulate components

of the components (besides what can be done by programming as exemplified by figure 9). For instance, it might be useful to generate an ADL such as the one on figure 7, and subsequently dynamically modify the description of the component application. Such tools could be integrated with computing portals and grid infrastructure middleware for resource brokering (ICENI [13], GridT [14], etc.), such as to build dedicated Problem Solving Environments [15].

We also investigate the following optimization: have functional method calls (either single or collective) bypass each inner composite component of a hierarchical component, so as to directly reach target primitive components – that are the only ones to serve functional service invocations. There is a non-trivial coherency problem to solve due to the concurrency of component management method calls (in particular, re-binding calls) towards encapsulating composite components.

Acknowledgments. This work was supported by the Incentive Concerted Action "GRID-RMI" (ACI GRID) of the French Ministry of Research and by the RNTL Arcad project funded by the French government.

References

1. Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., 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)
2. 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 Conference. (2000)
3. Denis, A., Pérez, C., Priol, T.: Achieving portable and efficient parallel corba objects. Concurrency and Computation: Practice and Experience (2003) To appear.
4. Denis, A., Pérez, C., Priol, T., Ribes, A.: Padico: A component-based software infrastructure for grid computing. In: 17th International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, IEEE Computer Society (2003)
5. Caromel, D., Klauser, W., Vayssiere, J.: Towards seamless computing and metacomputing in java. Concurrency Practice and Experience **10** (1998) 1043–1061
6. 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
7. Baduel, L., Baude, F., Caromel, D.: Efficient, flexible, and typed group communications in java. In: Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, ACM Press (2002) 28–36 ISBN 1-58113-559-8.
8. Bruneton, E., Coupaye, T., Stefani, J.: Recursive and dynamic software composition with sharing. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02) (2002)
9. Fractal. (`http://fractal.objectweb.org`)

10. ProActive web site. (http://www.inria.fr/oasis/ProActive/)
11. Sun Microsystems: Enterprise Java Beans Specification 2.0 (1998) `http://java.sun.com/products/ejb/docs.html`.
12. OMG: Corba 3.0 new components chapter (2001) Document ptc/2001-11-03.
13. Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J.: ICENI: Optimisation of Component Applications within a Grid Environment. Parallel Computing **28** (2002)
14. Godakhale, A., Natarajan, B.: Composing and Deploying Grid Middleware Web Services Using Model Driven Architecture. In: CoopIS/DOA/ODBASE. Number 2519 in LNCS (2002) 633–649
15. Rice, J., Boisvert, R.: From Scientific Libraries to Problem-Solving Environments. IEEE Computational Science and Engineering (1996) 44–53