Project no. FP6-034442

GridCOMP

Grid programming with COMPonents : an advanced component platform for an effective invisible grid

STREP Project

Advanced Grid Technologies, Systems and Services

D.CFI.03 – Architectural design of the component framework

Due date of deliverable: 01 June 2007

Actual submission date: 09 July 2007

Start date of project: 1 June 2006 **Duration**: 30 months

Organisation name of lead contractor for this deliverable: INRIA

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	PU

Keyword List: GCM, ProActive, reference implementation
Responsible Partner: Denis Caromel, INRIA

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
1	28-05-2007	Draft	Cédric DALMASSO
1.1	04-07-2007	Draft	Cédric DALMASSO
1.2	06-07-2007	Draft	Cédric DALMASSO
1.3	09-07-2007	Final	Cédric DALMASSO

Deliverable manager

- Denis Caromel, INRIA

List of Contributors

- Matthieu Morel
- Le Du
- Cédric Dalmasso

List of Evaluators

- Marco Danelutto
- Yongwei Wu

Summary

This document describes the architecture of the Component Framework Implementation (CFI) early prototype. The CFI is a previous GridCOMP deliverable, D.CFI.02.

Since the CFI is based on the ProActive middleware, we start by providing an overview of ProActive's architecture. We detail the model, the concept and the techniques used to implement them and finally, we describe the ProActive's features used in our implementation.

Then, in a second part, we explain how we use and extend this architecture in our implementation. We describe what we had to modify within ProActive and what we added on the top of it in order to achieve the goals specified for this implementation.

Table of Content

1 Introduction

This document describes the implementation architecture of the Grid Component Model (GCM) [GCM]. This model was defined in the Programming Model Institute by the CoreGrid community. The GridCOMP, EU funded, project will provide a reference implementation for the GCM. An early prototype was provided in a previous deliverable, the D.CFI.02. In this document, we describe the current architecture of this early prototype. Other GridCOMP deliverables explain how the mechanisms presented here are used. Particularly, in the WP3, D.NFCF.01 [NFC] document details how the non-functional features are implemented using the ProActive/GCM framework.

This early prototype is based on the ProActive middleware. It reuses and extends the principle implemented in ProActive. Therefore, we will first explain the concept used in the ProActive's architecture and after detail how we use them and extend them in the implementation of the GCM.

2 The ProActive middleware

ProActive is an open source Java library for Grid computing. It allows concurrent and parallel program and offers distributed and asynchronous communications, mobility, and a deployment framework. With a reduced set of primitives, ProActive provides an API allowing the development of parallel applications which may be deployed on distributed systems and on Grids.

2.1 Active objects model

ProActive is based on the concept of A*ctive Object* (AO), which can be seen as an entity with its own configurable activity.

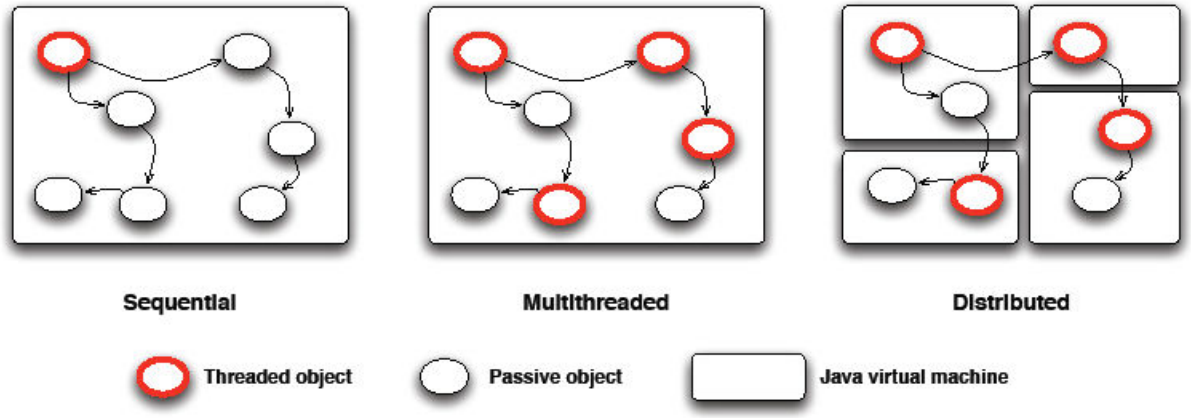

Figure 1 Seamless sequential to multithreaded to distributed objects

A distributed or concurrent application built using ProActive is composed of a number of active objects (**Figure 1**). 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 [CAR 93]. Method calls can be asynchronous only if they fulfil the following minimum conditions: reifiable return type and no declared exceptions in the method. A future is a placeholder for the result of an invocation, which is given as a result to the caller, and which is transparently updated when the result of the invocation is actually computed. This whole mechanism results in a data-based synchronization. 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, in order to ensure causal dependency.

Explicit message-passing based programming approaches were deliberately avoided: one aim of the library is to enforce code reuse by applying the remote method invocation pattern, instead of explicit message-passing.

2.2 The ProActive library: principles and architecture

The ProActive library implements the concept of active objects and provides a deployment framework in order to use the resources of a Grid.

ProActive is developed in Java in order to allow maximum portability. Moreover, ProActive only relies on standard APIs and does not use any operating-system specific routine, other than to run daemons or to interact with legacy applications. There are no modifications to the JVM or to the semantics of the Java language, and the bytecode of the application classes is never modified.

2.2.1 Implementation techniques

ProActive relies on extensible Meta-Object Protocol architecture (MOP), which uses reflective techniques in order to abstract the distribution layer, and to offer features such as asynchronism or group communications.

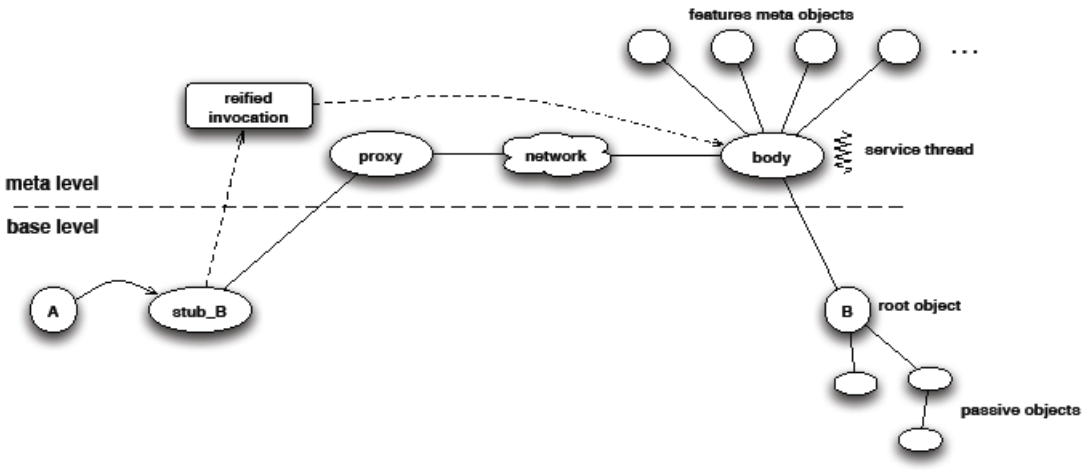

Figure 2 Meta-Object Architecture

The architecture of the MOP is presented in Figure 2. An active object is concretely built out of a root object (here of type B), with its graph of passive objects. A body object is attached to the root object, and this body references various meta-objects, with different roles and

providing features. An active object is always indirectly referenced through a proxy and a stub which is a sub-type of the root object. An invocation to the active object is actually an invocation on the stub object, which creates a reified representation of the invocation, with the method called and the parameters, and this reified object is given to the proxy object. The proxy transfers the reified invocation to the body, possibly through the network, and places the reified invocation in the request queue of the active object. There are adapters in each side of the network part, for the proxy and the body, allowing us to use several communication layers through the network. The request queue is one of the meta-objects referenced by the body. If the method returns a result, a future object is created and returned to the proxy, to the stub, then to the caller object.

The active object has its own activity thread, which is usually used to pick-up reified invocations from the request queue and serves them, i.e. execute them by reflection on the root object. Reification and interception of invocations, along with ProActive's customizable MOP architecture, provide both transparency and the ground for adaptation of non-functional features of active objects to fit various needs. It is possible to add custom meta-objects which may act upon the reified invocation, for instance for providing mobility features, or as we will see later, implement the GCM.

Active objects are instantiated using the ProActive API, by specifying the class of the root object, the instantiation parameters, and optional location information:

```
// instantiate active object of class B on node1
// (a possibly remote location)
B b = (B) ProActive.newActive(''B'', new Object[]
                                {aConstructorParameter}, node1);

// use active object as any object of type B
Result r = b.foo();

// possible wait-by-necessity
System.out.println(r.printResult());
```

2.2.2 Semantics of communications between Active Object

In ProActive, the distribution is transparent: invoking methods on remote objects does not require the developer to design remote objects with explicit mechanism allowing remote calls (like Remote interfaces in Java RMI). Therefore, the developer can concentrate on the business logic as the distribution is automatically handled and transparent. Moreover, the ProActive library preserves polymorphism on remote objects (through the reference stub, which is a subclass of the remote root object).

Communications between active objects are realized through method invocations, which are reified and passed as messages. These messages are serializable Java objects which may be compared to TCP packets. Indeed, one part of the message contains routing information towards the different elements of the library, and the other part contains the data to be communicated to the called object.

Although all communications proceed through method invocations, the communication semantics depends upon the signature of the method, and the resulting communication may not always be asynchronous. Three cases are possible: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

- *Synchronous invocation*:
    - the method return a non reifiable object: primitive type or final class:

```
public boolean foo()
```

    - the method declares throwing an exception:

```
public void bar() throws AnException
```

In this case, the caller thread is blocked until the reified invocation is effectively processed and the eventual result (or Exception) is returned. It is fundamental to keep this case in mind, because some APIs define methods which throw exceptions or return non-reifiable results. It is the case with some part of the GCM API.

- *One-way asynchronous invocation*: the method does not throw any exception and does not return any result:

```
public void gee()
```

The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object (in other words, once the rendez-vous is finished).

- *Asynchronous invocation with future result*: the return type is a reifiable type, and the method does not throw any exception:

```
public MyReifiableType baz()
```

In this case, a future object is returned and the caller continues its execution flow. The active object will process the reified invocation according to its serving policy, and the future object will then be updated with the value of the result of the method execution.

If an invocation from an object A on an active object B triggers another invocation on another active object C, the future result received by A may be updated with another future object. In that case, when the result is available from C, the future of B is automatically updated, and the future object in A is also updated with this result value, through a mechanism called automatic continuation.

2.2.3 Features of the library

As stated above, the MOP architecture of the ProActive library is flexible and configurable; it allows the addition of meta-objects for managing new required features. Moreover, the library also proposes a deployment framework, which allows the deployment of active objects on various infrastructures. The library may be represented in three layers: programming model, detailed in the previous section 2.1; non-functional features, such as fault-tolerance and security, and deployment facilities.

The deployment layer is detailed in the next section, and it allows the creation of remote active objects on various infrastructures. A peer-to-peer infrastructure is also available; it allows the acquisition of objects distributed within the infrastructure.

ProActive provides uses of several protocols in its communication layer between active objects: Java RMI as the default protocol, HTTP, tunnelled RMI. It is also possible to export active objects as web services, which can then be accessed using the standard SOAP protocol. A file transfer mechanism is also implemented, allowing the transfer of files between active objects; for instance to send large data input files or to retrieve results files [BAU 06].

2.2.4 Deployment framework

The deployment of Grid applications is often done manually, using remote shells for launching the various virtual machines or daemons on remote computers and clusters. The commoditization of resources through Grids and the increasing complexity of applications are making the task of deploying fundamental since it is harder to perform. ProActive succeeds in completely avoiding scripts for configuration, getting computing resources, etc. It provides, as a key approach to the deployment problem, an abstraction from the source code so as to gain in flexibility. Now, we describe the fundamental principles of the deployment framework, and more information and examples are available from the official ProActive documentation [PRO].

Principles

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

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

The objective is to deploy any application anywhere without changing the source code. Deployment sites are called nodes, and correspond for ProActive to JVMs which contain active objects. A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. The two following requirements are needed to abstract the underlying execution platform and keep the source code independent from deployment:

- 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.

XML deployment descriptors

To answer these requirements, the deployment framework in ProActive relies on XML descriptors. These descriptors introduce the notion of Virtual Node (VN):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,

- a VN, after activation, is mapped to either one or a set of actual ProActive nodes, following the mapping defined in an XML descriptor file.

A virtual node represents 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 mapping between virtual nodes and nodes created by the deployment. This mapping is specified in the deployment descriptor. There is no automatic mapping between virtual nodes and active objects: the active objects are deployed by the application on the infrastructure nodes mapped by a virtual node. By definition, the following operations can be configured in the deployment descriptor:

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

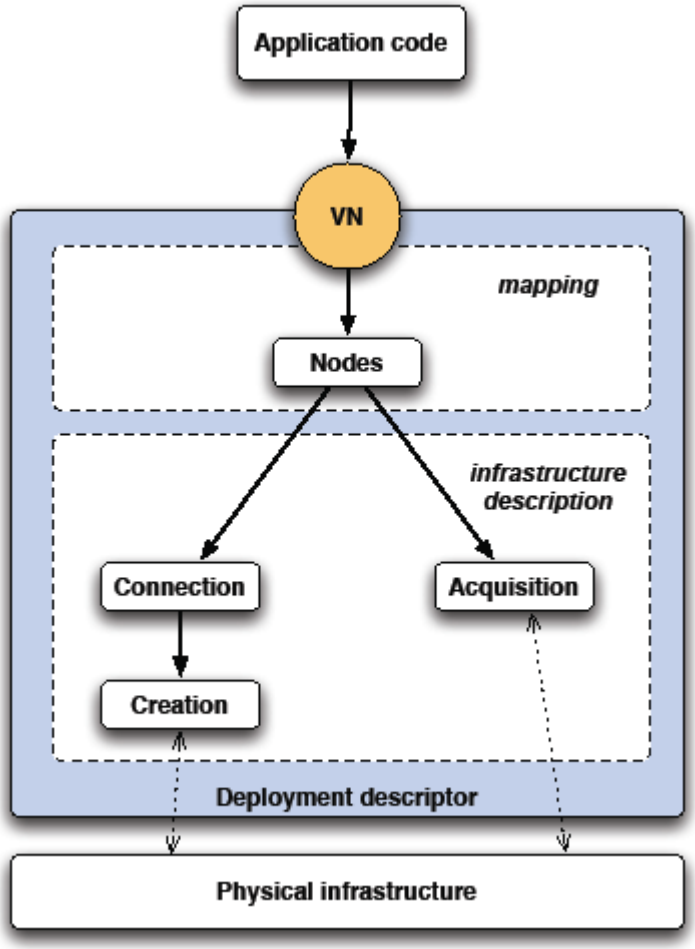

Figure 3 The deployment framework in ProActive.

Figure 3 summarizes the deployment framework provided in the ProActive middleware. Deployment descriptors can be separated in two parts: mapping and infrastructure. The VN, which is the deployment abstraction for applications, is mapped to nodes in the deployment descriptors, and nodes are mapped to physical resources, i.e. to the infrastructure.

Retrieval of resources

In the context of the ProActive middleware, nodes designate physical resources of a physical infrastructure. They can be created or acquired. The deployment framework is responsible for providing the nodes mapped to the virtual nodes used by the application. Nodes may be created using remote connection and creation protocols. Nodes may also be acquired through lookup protocols, which enable access to the ProActive peer-to-peer infrastructure, for instance.

Creation-based deployment: Machine names, connection and creation protocols are strictly separated from the application code, and ProActive deployment descriptors provide the ability to create remote nodes (remote JVMs). For instance, deployment descriptors are able to use various protocols:

- local,
- ssh, gsissh, rsh, rlogin,
- lsf, pbs, sun grid engine, oar, prun,
- globus (GT2, GT3 and GT4), unicore, glite, arc (nordugrid).

Deployment descriptors allow combining these protocols in order to create remote JVMs, e.g. log on a remote cluster frontend with SSH, and then use pbs to book cluster nodes to create JVMs on each. In addition, the localJVM process allows the JVM creation. It is possible to specify the *classpath*, the Java install path, and all JVM arguments. It is in this process that the deployer specifies which transport layer the ProActive node uses. For the moment, ProActive supports RMI, HTTP, RMIssh, Ibis and SOAP as transport layers.

Acquisition-based deployment: The main goal of the peer-to-peer (P2P) infrastructure is to provide a new way to build and use Grids. The infrastructure allows applications to transparently and easily obtain computational resources from Grids composed of both clusters and desktop machines. The burden of application deployment is eased by a seamless link between applications and the infrastructure. This link allows applications to communicate, and to manage the resources volatility.

2.3 Conclusion

In this first part, we introduced the ProActive grid middleware and we described the architecture of its current implementation. In the following section we will explain how we used the ProActive framework to implement our early prototype of component framework.

3 Architecture of the CFI implementation

In this section, we describe the architecture of the CFI, which implement the GCM. We named this implementation ProActive/GCM. This early prototype is based on the ProActive middleware and extends its architecture.

3.1 Design goals

This framework was designed following these main objectives:

1. Follow the GCM specification.
2. Base the implementation on the concept of active objects. The components in this framework are implemented as active objects, and as a consequence benefit from the properties of the active object model.
3. Leverage the ProActive library by proposing a new programming model which may be used to assemble and deploy active objects. Components in the ProActive library therefore also benefit from the underlying features of the library as described in section 2.2.3.
4. Provide a customizable framework, which may be adapted by the addition of non functional controllers and interceptors for specific needs, and where the activity of the components is also customizable.

We also propose some optimizations; they are achieved to the expense of a trade-off between dynamicity (the possibility to dynamically reconfigure the applications, or parts of the applications) and efficiency (direct or multithreaded invocations).

3.2 An architecture based on ProActive's Meta-Object Protocol

The ProActive/GCM framework is an implementation of the GCM specification which extends the Fractal 2 specification [FRAa]. It follows the general model described in the GCM specification and implements the GCM Java API.

Our implementation of GCM relies on ProActive's Meta-Object Protocol (MOP) architecture.

3.2.1 Component instance

A ProActive/GCM component is an active object. The implementation of a ProActive/GCM component therefore follows the general architecture represented in Figure 2. As we stated in the presentation of the ProActive library, the reflective framework may be customized by adding or specializing meta-objects. This allowed us to implement GCM components using a reflective framework.

A component is instantiated using the GCM API (GCM is based on Fractal API [FRAa], thus in the following examples this API will be heavily used):

```
// get bootstrap component
Component boot = Fractal.getBootstrapComponent();
// get type factory
TypeFactory tf = Fractal.getTypeFactory(boot);
// get generic component factory
GenericFactory gf = Fractal.getGenericFactory(boot);
// define component type
ComponentType type = tf.createFcType(....);
// define controller description
ControllerDescription controllerDesc = new ControllerDescription(name,
```

```
     hierarchicalType);
// define content description
ContentDescription contentDesc = new
ContentDescription(implementationClass,
     constructorParameters);
// instantiate component
Component c = gf.newFcInstance(type, controllerDesc, contentDesc);
```

The bootstrap component is retrieved by checking the fractal.provider java property (in our implementation, `org.objectweb.proactive.core.component.Fractive`). The controller part of the component is described in a `ControllerDescription` object. The content part of the component is described in a `ContentDescription` object.

The instance of a component is represented in Figure 4. The `newFcInstance` method on the component factory returns a Component object. It is a remote reference of type `Component` on the active object which implements the component.

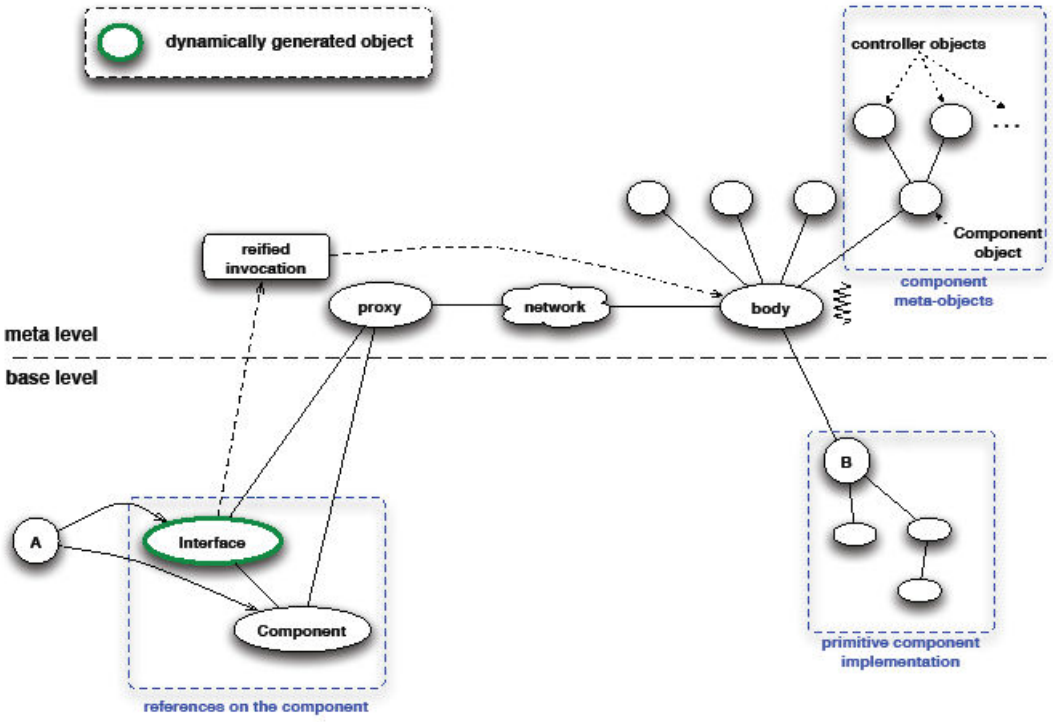

Figure 4 ProActive Meta-Object architecture for primitive components

Before describing the architecture of a component in the ProActive library, we first need to clarify the terminology concerning the typing, between objects and components. In the Java language, which follows the object paradigm, the live entities are objects. An object is an instance of a class. The services offered by the class are defined by the methods of this class. In the GCM, which follows the component paradigm, the live entities are instances of components. The services offered by the component are defined by its server interfaces. These server interfaces themselves define methods, which are the actual services.

As a consequence, in the object paradigm, an instantiation returns an object of a type compatible with the specified class, whereas in the component paradigm, an instantiation returns a component of type compatible with the specified component type. In the GCM Java API, a reference on a component is a reference on an object of type Component.

Figure 4 represents an instance of a ProActive/GCM primitive component and Figure **5** represents a composite one.

The design of the implementation of ProActive/GCM components relies on the general design of active objects represented in Figure 2. It however exhibits specificities. First of all, a reference on a component from an object A is a reference on a `Component` object called the representative, which we can clearly see in the bottom left-hand corner of the figure. This Component object acts as a stub in the standard ProActive architecture, although for performance reasons, a smart proxy pattern is implemented so that common operations, such as getting a reference on a component interface, are performed locally. Using the services of a component implies getting a reference on a given named interface (using the `getFcInterface` method), then invoking methods on this interface. The instance of the `Component` object holds references on local representatives of the functional and non functional interfaces. These representatives act as stub objects, as they reify invocations and transmit these reified invocations to the proxy. The interfaces representatives are generated dynamically at the creation of the component, or when retrieving a reference on this component through a lookup mechanism. For the sake of clarity, only one of these `Interface` object is represented on this figure, although all functional and non functional interfaces are dynamically created locally when creating the reference to the component.

The controller part of the component is implemented as meta-objects as can be seen in the top right-hand corner of the figures. These meta-objects implement controllers, in particular the basic controllers (binding, lifecycle etc...).

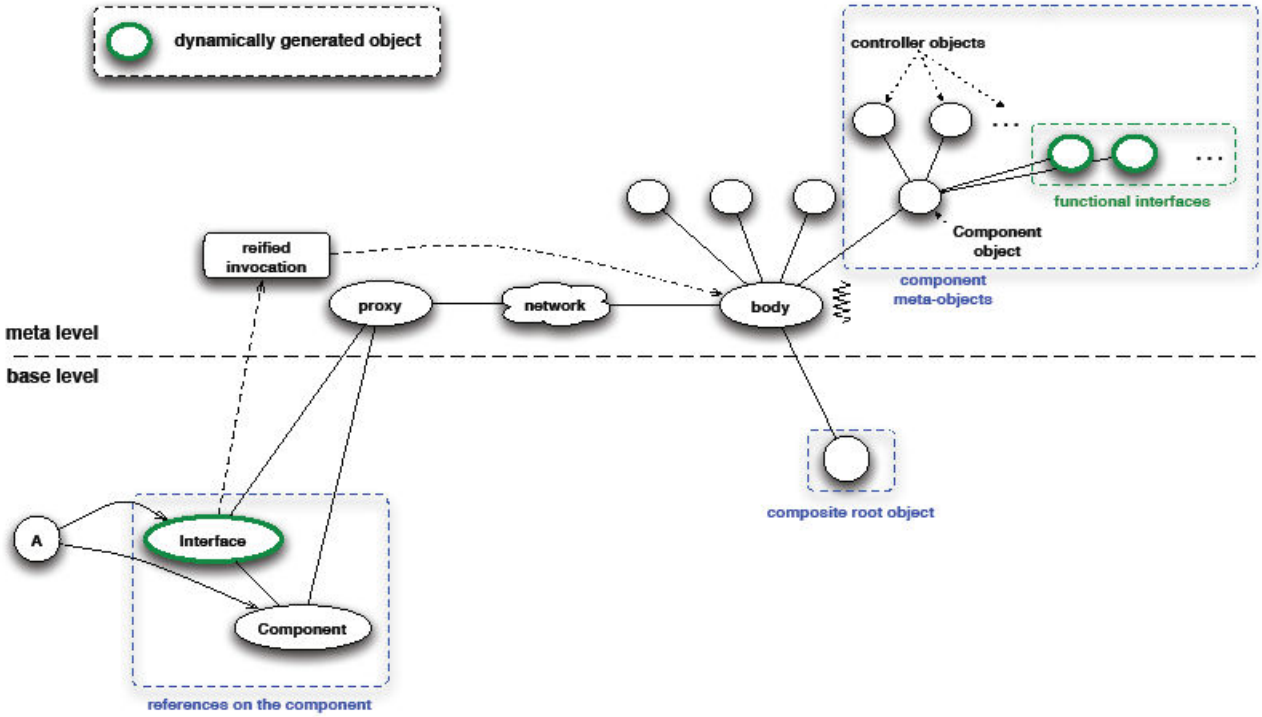

Figure 5 ProActive Meta-Object architecture for composite components

3.2.1.1 Primitive components

In a primitive component, the content of the component corresponds to an implementation class, which in ProActive is the root object of the active object, as represented on the bottom right-hand corner of the Figure 4. Following the GCM specification, the primitive class may have to implement some callback interfaces such as `BindingController` or `AttributeController`, which are invoked from the meta-level for performing operations which are dependent on the applicative implementation code.

3.2.1.2 Composite components

Figure **5** represents an instance of a composite component. A composite component is a structuring component which does not have any business code. Hence the empty composite object as the root of the active object. However, a composite component still offers and requires functional services, and the interface objects corresponding to these services are implemented as meta-objects, as represented on the top right-hand corner of the figure. They may represent internal client interfaces or external client interfaces. A composite component also offers a `ContentController` interface and implementation as a meta-object, for controlling the components it may contain.

3.2.2 Configuration of controllers

The control part of the component is fully customizable, and the configuration is specified in an XML file, which specifies which control interfaces are offered, and which control classes implement the control interfaces. The default configuration file is provided in the Appendix A. We can see the standard required controllers for binding, content, name, super ... The binding controller is actually only instantiated in case of client interfaces, and the content controller is only instantiated for composite components. The `ComponentParametersController` allows setting specific parameters for this implementation. Some other controllers are related to the features offered by this implementation: migration, management of gathercast and multicast interfaces.

For instance with the `BindingController`, we can see that for each controller we define the java interface and its implementation class. The section 3.2.4 shows how we can use this configuration file to use controller as interceptor.

```
        <controller>
              <interface>
  org.objectweb.proactive.core.component.controller.ProActiveBindingCon
troller
              </interface>
              <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveBindingCon
trollerImpl
              </implementation>
        </controller>
```

3.2.3 Lifecycle

The lifecycle of components is implemented by customizing the activity of the active objects.

ProActive offers the possibility to customize the activity of an active object; this is actually a fundamental feature of the library, as it allows to fully specify the behaviour of active objects. In the context of components, we distinguish the non-functional activity from the functional activity. The non-functional activity corresponds to the management of component requests when the lifecycle of the component is stopped (i.e. only control requests). The functional activity is encapsulated and starts when the lifecycle is started. This is illustrated in Figure 6. The default behaviour is to serve all control requests in a FIFO order until the component is started using the `lifecycle-controller`. Then, a component serves all requests, control and functional, in a FIFO order, until the lifecycle is stopped. The functional activity is encapsulated in the component activity.

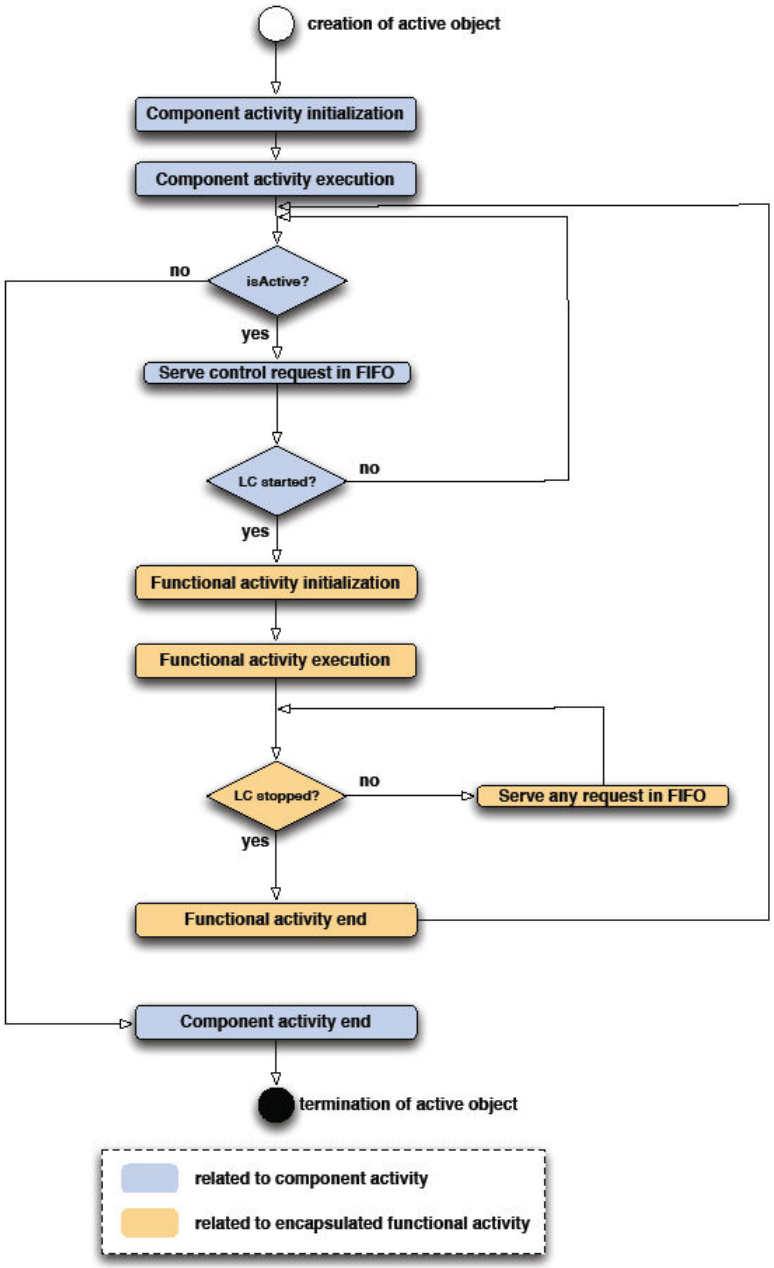

Figure 6 Default encapsulation of functional activity inside component activity

It is possible to fully customize this behaviour. First, the functional activity may be customized by implementing the `InitActive`, `RunActive` and `EndActive` interfaces.

Two conditions must be respected though, for a smooth integration with the component lifecycle:

1. The control of the request queue must use the `org.objectweb.proactive.Service` class.

2. The functional activity must loop on the `body.isActive()` condition (this is not compulsory, but it allows to automatically end the functional activity when the lifecycle of the component is stopped. It may also be managed with a custom filter on the request queue).

By default, when the lifecycle is started, the functional activity is initialized, run, then ended when the `!isActive()` condition is true (this method is overridden if the active object is a component, so that this condition is valid when the lifecycle is set to stopped).

Second, the component activity itself may be customized, by implementing the `ComponentInitActive`, `ComponentRunActive` and `ComponentEndActive` java interfaces.

3.2.4 Interception mechanism

The GCM specification states that a component controller can intercept incoming and outgoing operation invocations targeting or originating from the component's subcomponents. This feature is provided in the ProActive/GCM implementation, and it allows an interception at the meta-level, of reified invocations, with configurable pre and post method processing. It is an easy way of providing AOP-like features, in order to deal notably with non functional concerns. Interceptors may intercept incoming and outgoing invocations, and they are sequentially combined. An interceptor is a component controller with some additional implemented interface allowing the definition of actions to do before and/or after a communication. These capabilities could be used for example in a non-functional controller wanting to react in function of the communication activities (time to serve request, number of served request …).

An input interceptor is a controller which must implement the `org.objectweb.proactive.core.component.interception.InputInterceptor` interface, which defines the following methods:

```
public void beforeInputMethodInvocation(MethodCall methodCall);
public void afterInputMethodInvocation(MethodCall methodCall);
```

The `MethodCall` object represents the reified invocation in the ProActive library. Similarly, an output interceptor must implement the `org.objectweb.proactive.core.component.interception.OutputInterceptor` interface, which defines the following methods:

```
public void beforeOutputMethodInvocation(MethodCall methodCall);
public void afterOutputMethodInvocation(MethodCall methodCall);
```

The *input interception* mechanism occurs at the service of the request in the membrane: the reified request is delegated to the input controllers before and after the method is processed.

The *output interception* mechanism occurs in the interface representative (whose code is dynamically generated as showed in the bottom left-hand corner Figure 4 and Figure **5**) when the invocation is reified: before and after transferring the invocation to the connected component, the reified request is delegated to the output interceptors. The output interception is realized by replacing, during a binding operation, the server interface representative by a server interface representative of the same type, containing the interception code.

Interceptors are configured in the controllers XML configuration file, by simply adding `input-interceptor="true"` or/and `output-interceptor="true"` as attributes of the controller element in the definition of a controller (provided of course the specified interceptor is an input or/and output interceptor). For example a controller that would be an input interceptor and an output interceptor would be defined as follows:

```
<controller input-interceptor="true" output-interceptor="true">
  <interface>InterceptorControllerInterface</interface>
  <implementation>ControllerImplementation</implementation>
</controller>
```

For input interceptors, the `beforeInputMethodInvocation` method is called sequentially for each controller in the order in which they are defined in the controllers configuration file. The `afterInputMethodInvocation` method is called sequentially for each controller in the *reverse order* they are defined in the controllers configuration file. For instance, in the following controller configuration file, the list of input interceptors declares first, `InputInterceptor1`, and second, `InputInterceptor2;` then, an invocation on a server interface will follow the path described in Figure 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="component-config.xsd"
  name="defaultConfiguration">
  <!-- … other controllers -->
  <!-- input interceptors -->
  <controllers>
   <controller input-interceptor="true">
     <interface> InputInterceptor1</interface>
     <implementation> InputInterceptor1Implementation</implementation>
   </controller>
   <controller input-interceptor="true">
     <interface> InputInterceptor2</interface>
     <implementation> InputInterceptor2Implementation</implementation>
   </controller>
  </controllers>
</componentConfiguration>
```

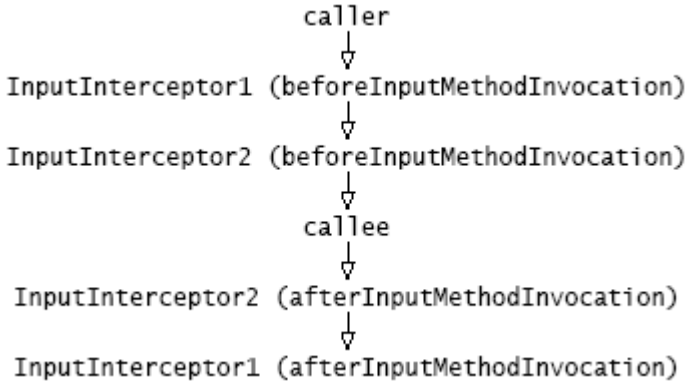

Figure 7 Execution sequence of an input interception

For output interceptors, the `beforeOutputMethodInvocation` method is called sequentially for each controller in the order they are defined in the controllers configuration file. The `afterOutputMethodInvocation` method is called sequentially for each controller in the *reverse order* they are defined in the controllers configuration file. For instance, in the following controller configuration file, the list of output interceptors declares first `OutputInterceptor1` and second `OutputInterceptor2`; then, an invocation on a client interface will follow the path described in Figure 8.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="component-config.xsd"
  name="defaultConfiguration">
  <!-- … other controllers -->
  <!-- output interceptors -->
  <controllers>
   <controller output-interceptor="true">
     <interface>OutputInterceptor1</interface>
     <implementation>OutputInterceptor1Implementation</implementation>
   </controller>
   <controller output-interceptor="true">
     <interface>OutputInterceptor2</interface>
     <implementation>OutputInterceptor2Implementation</implementation>
   </controller>
  </controllers>
</componentConfiguration>
```

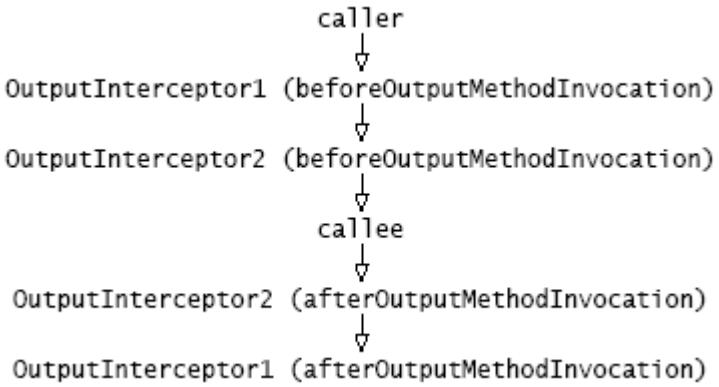

Figure 8 Execution sequence of an output interception

An interceptor being a controller, it must follow the rules for the creation of a custom controller (in particular, extend `AbstractProActiveController`). Input interceptors and output interceptors must implement respectively the interfaces `InputInterceptor` and `OutputInterceptor` respectively, which declare interception methods (pre/post interception) that have to be implemented.

Here is a simple example of an input interceptor:

```
public class MyInputInterceptor extends AbstractProActiveController
 implements InputInterceptor, MyController {
  public MyInputInterceptor(Component owner) {
    super(owner);
  }
  // some init code
  ...
  // foo is defined in the MyController interface
  public void foo() {
    // foo implementation
  }
  public void afterInputMethodInvocation(MethodCall methodCall) {
    System.out.println("post processing an intercepted an incoming
functional invocation");
    // interception code
  }
  public void beforeInputMethodInvocation(MethodCall methodCall) {
    System.out.println("pre processing an intercepted an incoming
functional invocation");
    // interception code
 }
}
```

The controller is included in the configuration of a given component as follows:

```
<componentConfiguration>
 <controllers>
    ....
   <controller input-interceptor="true">
     <interface>MyController</interface>
     <implementation>MyInputInterceptor</implementation>
   </controller>
   ...
```

3.2.5 Communications

Communications between components in ProActive/GCM occur through interface references, and rely on the standard ProActive communication mechanism. They may therefore use any underlying protocol supported by ProActive (RMI, RMIssh, http...), and the semantics of invocations are kept, which means that some conditions must be respected for an invocation to be asynchronous. In particular, if the invoked method throws any exception, the invocation is synchronous.

The section 3.3 describes precisely the architecture for collective interface. As well as standard communication, the GCM [GCM] allows data, stream and event ports to be used in component interaction. For the moment, there are requirements for this kind of communication, consequently we are eventually going to provide the stream/data/event ports. We will consider results from other projects, such as the Dream project [DRE] in the Fractal

community that provides components implementing various communication paradigms including stream, event, etc.

3.2.5.1 Optimization with short cuts ('shortcut')

Communications between components in a hierarchical model may involve crossing several membranes of enclosing composite components, and therefore paying the cost of several indirections. If the invocations are not intercepted in the membranes, then it is possible to optimize the communication path by communicating directly from a caller component to a callee component, avoiding indirections in the membranes.

We provide a short cut mechanism for distributed components, and the implementation of this mechanism relies on a tensioning technique: the first invocation determines the short cut path, and then the following invocations will use this short cut path. This mechanism requires composite components to be passive components: objects that augment a root object with a MOP, but do not have any request queue. As a consequence, the rendez-vous of the communication between a client and a server interface, which guarantees causally ordered communications, does not end until the effective server interface has been reached (and the calling thread has returned).

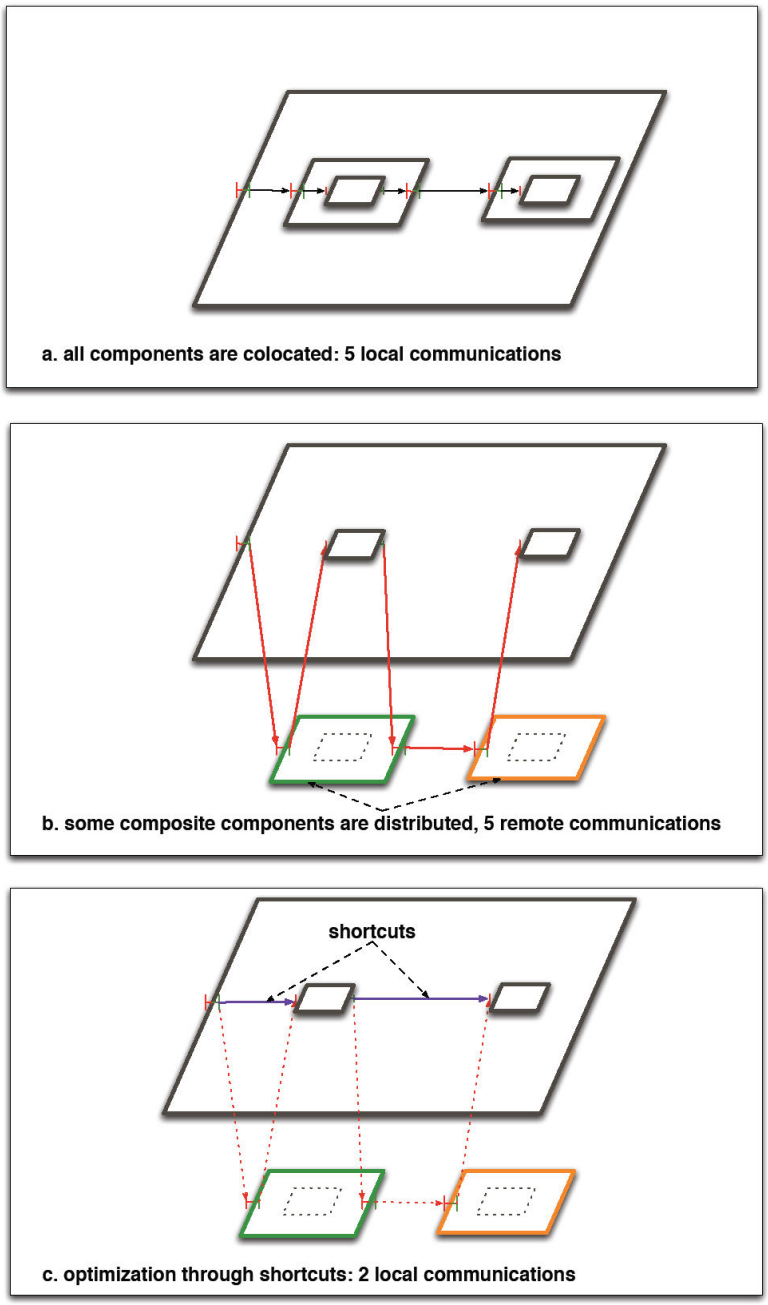

Figure 9 Using short cuts for minimizing remote communications

For instance, in Figure 9, a simple component system, which consists of a composite containing two wrapped primitive components, is represented with different distributions of the components. In *a*, all components are located in the same JVM, therefore all communications are local communications. If the wrapping composites are distributed on different remote JVMs, all communications are remote because they have to cross composite enclosing components. The short cut optimization is a simple bypass of the wrapper components, which results in 2 local communications for the sole functional interface. The shortcut mechanism handle also other situation, in fact all communication through synchronous composite component could be bypassed.

Short cuts are available when composite components are synchronous components (this does not break the GCM model, as composite components are structural components unless that composite used autonomic non functional features). Components can be specified as

synchronous in the `ControllerDescription` object that is passed to the component factory:

```
ControllerDescription controllerDescription =
       new ControllerDescription("name", Constants.COMPOSITE,
                                 Constants.SYNCHRONOUS);
```

When the system property `proactive.components.use_shortcuts` is set to true, the component system automatically establishes short cuts between components whenever possible.

3.3 Mechanism and implementation of collective interfaces

In order to provide facilities for parallel programming, GCM defines collective interfaces. To sum it up, the idea is to introduce multicast and gathercast interfaces: multicast interfaces are used for parallel invocations and data redistribution, and gathercast interfaces are used for synchronization and data gathering. The configuration of the collective interfaces policies uses annotations in Java interfaces.

The signatures of methods of client and server interfaces are different when using collective interfaces and different dispatch or gather mode. For list parameters and return types, the possibilities in our implementation are summarized in Figure 10. Broadcast mode is not yet supported for the redistribution of results in gathercast interfaces.

	client interface	*server interface*
multicast	List<A> foo (List<B>)	A foo (List<B>) (broadcast mode) A foo (B) (scatter mode)
gathercast	A bar (B) (scatter mode) ~~List<A> bar (B) (broadcast mode)~~	List<A> bar (List<B>)

Figure 10 Adaptation of method signatures, with list parameters or return types, between client and server interfaces for collective interfaces.

The framework provides transparent adaptation of method invocations and distribution of parameters, through proxies and controllers. Compatibility of client and server interfaces is checked at runtime, although this could be checked at design-time using assembly tools.

3.3.1 Multicast interfaces

Our implementation of multicast interfaces does not currently provide dynamic dispatch but the distribution policies are customizable.

The implementation of multicast interfaces relies on two principles: first, reuse the existing mechanism for typed group communications in ProActive, and second use a delegation mechanism for adapting the signatures of the interfaces. Therefore, two group proxies are used for a multicast invocation: the first proxy corresponds to the signature of the client

interface, and the second one to the signature of the server interfaces. Bindings are transparently handled so that the client component receives a reference on a group proxy of the type of the client interface.

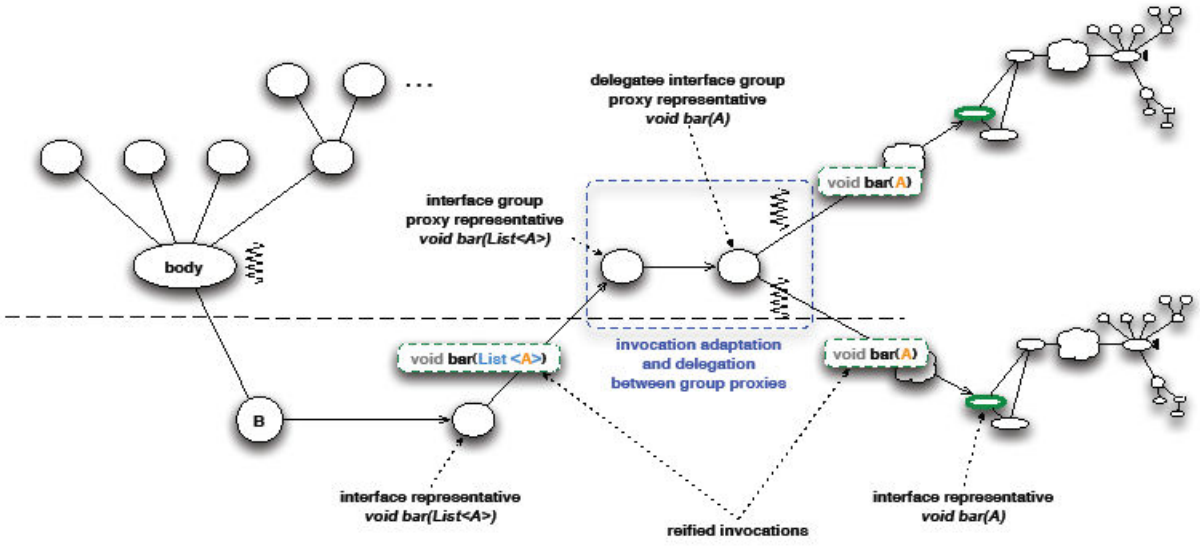

Figure 11 Adaptation and delegation mechanism for multicast invocations

This mechanism is illustrated in Figure 11, which corresponds to the design represented in Figure 12.

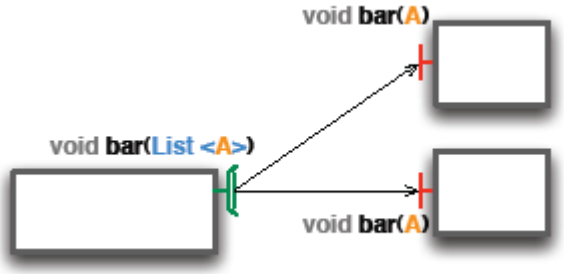

Figure 12 An example of multicast interfaces: the signature of an invoked method is exposed, and in this case exhibits a scattering behaviour for the parameters

When an invocation is performed, a reified invocation is first created (here on method `void bar(List<A>)`), given to the first group proxy, which delegates it to a second proxy of the type of the server interfaces (for invocations on method `void bar(A)`). Parameters are then automatically distributed according to the distribution policy specified as an annotation, and the second proxy transfers the new reified invocations to connected server interfaces in a parallel manner (using the standard multithreading mechanism of ProActive typed groups). This delegation and adaptation process between group proxies is implemented by extending the standard group proxy, the `ProxyForGroup` class, into the `ProxyForComponentInterfaceGroup`.

3.3.1.1 Configuration

The distribution of parameters in our framework is specified in the definition of the multicast interface, using annotations. Elements of a multicast interface which can be annotated are:

interface, methods and parameters. The different distribution modes are explained later. The examples in this section all specify broadcast as the distribution mode.

Interface annotations

A distribution mode declared at the level of the interface defines the distribution mode for all parameters of all methods of this interface, but may be overridden by a distribution mode declared at the level of a method or of a parameter. The annotation for declaring distribution policies at level of an interface is `@org.objectweb.proactive.core.component.type.annotations.multicast.ClassDispatchMetadata` and is used as follows:

```
@ClassDispatchMetadata(
  mode = @ParamDispatchMetadata(mode = ParamDispatchMode.BROADCAST)
)
interface MyMulticastItf {
  public void foo(List<T> parameters);
}
```

Method annotations

A distribution mode declared at the level of a method defines the distribution mode for all parameters of this method, but may be overridden at the level of each individual parameter. The annotation for declaring distribution policies at level of a method is `@org.objectweb.proactive.core.component.type.annotations.multicast.MethodDispatchMetadata` and is used as follows:

```
@MethodDispatchMetadata(
  mode = @ParamDispatchMetadata(mode = ParamDispatchMode.BROADCAST)
)
public void foo(List<T> parameters);
```

Parameter annotations

The annotation for declaring distribution policies at level of a parameter is `@org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatchMetadata` and is used as follows:

```
public void foo(
  @ParamDispatchMetadata(mode = ParamDispatchMode.BROADCAST)
  List<T> parameters);
```

For each method invoked and returning a result of type T, a multicast invocation returns an aggregation of the results: a List<T>. There is a type conversion, from return type T in a method of the server interface, to return type List<T> in the corresponding method of the multicast interface. The framework transparently handles the type conversion between return types.

Available distribution policies

Three modes of distribution of parameters are provided by default, and define distribution policies for lists of parameters:

- **BROADCAST** copies a list of parameters and sends a copy to each connected server interface.

- **ONE-TO-ONE** sends the *ith* parameter to the connected server interface of index *i*. This implies that the number of elements in the annotated list must be equal to the number of connected server interfaces.

- **ROUND-ROBIN** distributes each element of the list parameter in a round-robin fashion to the connected server interfaces. For *n* elements in the list parameter, *n* method calls are made.

It is also possible to define custom distributions by specifying the distribution algorithm in a class. This class needs to implement the `org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatch` interface, thereby defining the distribution algorithm which will be used during the dispatch phase. There are only three methods to implement:

```
public List<Object> dispatch (Object inputParameter,
  int nbOutputReceivers) throws ParameterDispatchException;

public int expectedDispatchSize (Object inputParameter,
  int nbOutputReceivers) throws ParameterDispatchException;

public boolean match (Type clientSideInputParameter,
  Type serverSideInputParameter) throws ParameterDispatchException;
```

Then the custom dispatch mode is used as follows:

```
@ParamDispatchMetadata(mode = ParamDispatchMode.CUSTOM,
  customMode = CustomParametersDispatch.class))
```

3.3.2 Gathercast interfaces

The implementation of gathercast interfaces in our framework is restricted to the management of a basic synchronization. Synchronization policy is not configurable, except for a timeout which can be specified if the method returns a result. Data redistribution policies for results are not configurable and the redistribution of results occurs in a one-to-one manner to the client interfaces.

Bindings to gathercast interfaces are bi-directional, since gathering operations require knowledge of the participants, which means that the server gathercast interface holds a reference to its clients. This is used for synchronization: once an invocation on a given method `foo1` comes from a client interface, the gathercast interface will create the corresponding request to be processed by the server component until all clients have sent an invocation on this method `foo1`. Until this condition is reached, the requests are queued in a special queue.

When the reified invocation on method `foo1` from the last connected client is served, the synchronization condition is reached, and a new reified invocation is created by gathering all parameters from all client invocations. The new reified invocation is then served by the server component.

The data structure representing the queues of requests (reified invocations) is illustrated in Figure 13. In the figure, we can see the enqueued requests for one gathercast interface (`gathercastItf1`) and two different methods. Suppose we have three clients, and *Ri* is an incoming request from client *i*. In the case of `foo1`, when the request on this method coming from client *3* will be served, then a new request will be created and served by the component, and the queue will be emptied of the corresponding requests (*R1*, *R2* and *R3* corresponding at the first line of the box in column '*request from client*' and line '*foo1'* in the Figure 13). We can also observe that client *1* invoked `foo1` twice, but the mechanism waits for the first queue to be full until processing any other queue, even though they are full. This is a way to guarantee causal dependency.

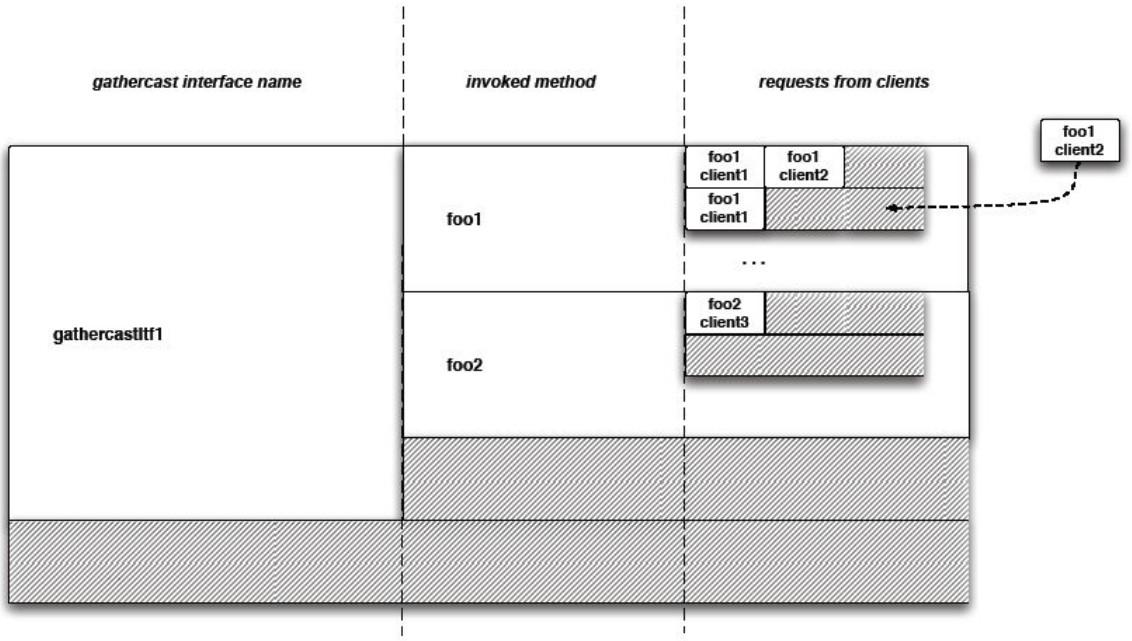

Figure 13 Data structure for the buffering of requests in gathercast interfaces

3.3.2.1 Asynchronism and management of futures

One fundamental feature of the ProActive/GCM is the asynchronism of method invocations: we want to preserve it in the context of gathercast interfaces, not only between client and server gathercast interface, but also for the transformed invocation in the gathercast interface.

When the invoked method returns void, there is no problem as this is considered as a one way invocation in ProActive, no future result is expected. If the invoked method returns a result however, the method returns a future, although the invocation has not been processed yet (an invocation on a gathercast interface will not proceed until all client interfaces invoked the same method). We faced a complex problem: how to return and update futures of client invocations on gathercast interfaces? We considered two strategies. The first one was to customize the request queue so that a local data structure (similar to the one described in Figure 13) would handle the incoming requests for gathercast interfaces. A second option was to use a dedicated tier active object for handling futures.

As we did not want to intervene in the core of the ProActive library by modifying the request queue, we selected and implemented the second option. The mechanism is illustrated in Figure 14. One futures handler active object is created for each gathercast request to be

processed. It has a special activity, which only serves distribute requests once it has received the `setFutureResult` request.

When a request from a client is served by the gathercast interface, it is enqueued in the queue data structure, and the result which is returned is the result of the invocation of the distribute method (with an index) on the futures handler object. This result is therefore a future itself.

When all clients have invoked the same method on the gathercast interface, a new request is built and served, which leads to an invocation which is performed either on the base object if the component is primitive, or on another connected interface if the component is composite. The result of this invocation is sent to the futures handler object, by invoking the `setFutureResult` method. The futures handler will then block until the result value is available. Then the distribute methods are served and the values of the futures received by the clients are updated.

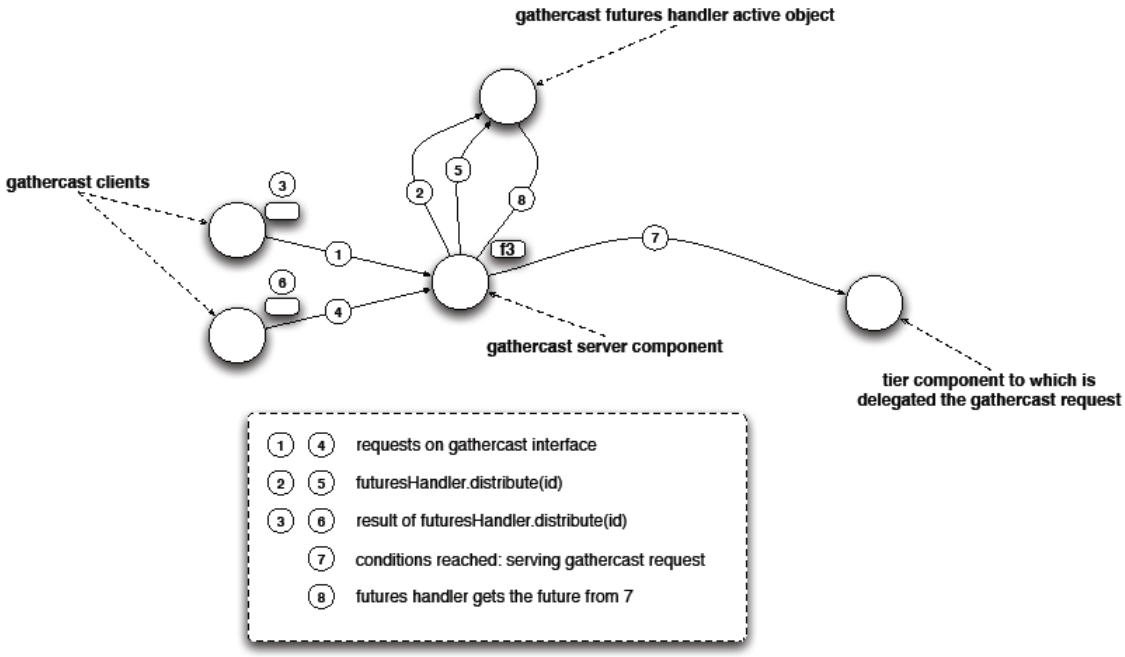

Figure 14 Management of futures for gathercast invocations

Although this mechanism fulfils its role using the standard mechanism of the library, we observed that it does not scale very well: one active object for managing futures is created for each gathercast request, and even though we implemented a pool of active objects, there are too many active objects created the gathercast interface is stressed. Therefore, the first approach described above should be preferred in the future.

3.3.2.2 Timeout

It is possible to specify a timeout, which corresponds to the maximum amount of time between the moment the first invocation of a client interface is processed by the gathercast interface, and the moment the invocation of the last client interface is processed. Indeed, the gathercast interface will not forward a transformed invocation until all invocations of all client interfaces are processed by this gathercast interface. Timeouts for gathercast invocations are specified by an annotation on the method subject to the timeout, the value of the timeout is specified in milliseconds:

```
@org.objectweb.proactive.core.component.type.annotations.gathercast.Met
hodSynchro(timeout = 20)
```

If a timeout is reached before a gathercast interface could gather and process all incoming requests, a `org.objectweb.proactive.core.component.exceptions.GathercastTimeoutException` is returned to each client participating in the invocation. This exception is a runtime exception. Timeouts are only applicable to methods which return a non-void value: there is no simple way otherwise to inform the client that the timeout has been reached: the client would need to provide a callback interface, which does not fit well with a simple invocation-based programming model.

3.4 Deployment

The deployment process is based on both the Fractal ADL [FRAb] capabilities and the ProActive deployment framework. A component system is usually described using an ADL, and the location of the components is specified in the ADL using the virtual node abstraction. Virtual nodes are then mapped to the physical infrastructure by using the standard ProActive deployment mechanism as described in 2.2.4.

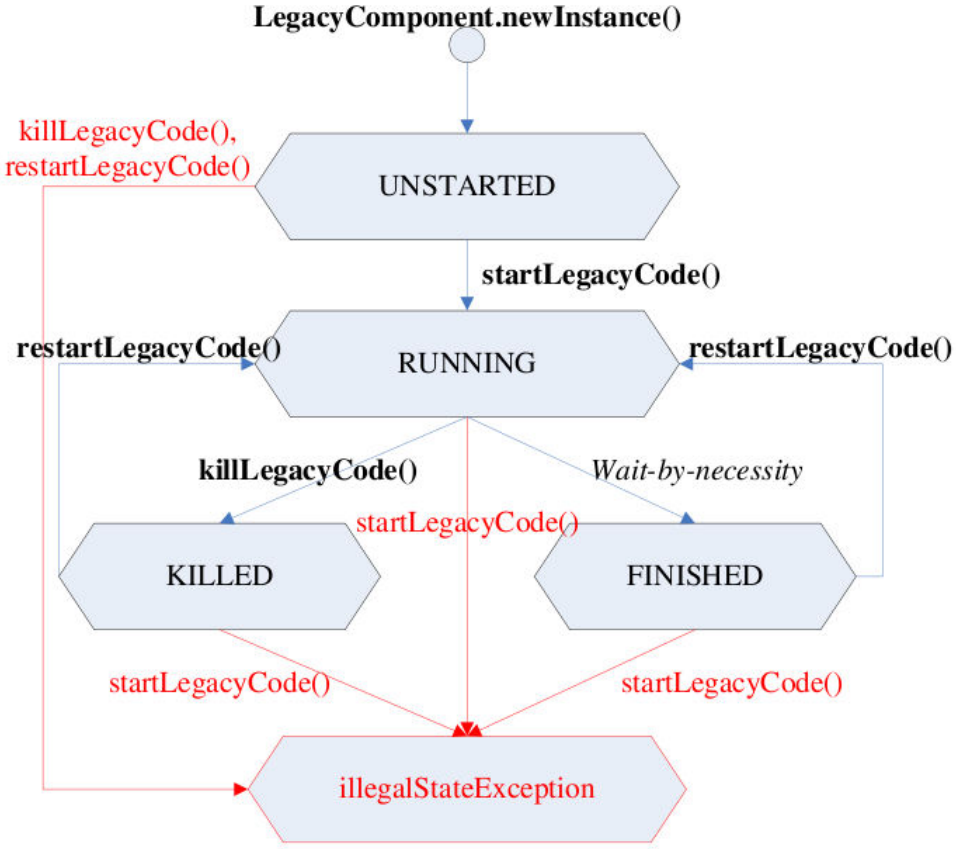

Figure 15 The execution status transition of the legacy code

3.5 Legacy code wrapping

In order to take into account the legacy code specificities, we define some specific structures and tags in the ADL. We also design some methods for turning those codes into components, which includes some standard API to manipulate and control the code. Using these methods, we can describe the legacy code in the ADL and wrap it to Component. With the standard API, we can control the running process of the legacy code.

There are many kinds of legacy codes, such as MPI Program or Executable Program running on the single computer. However, almost all the legacy codes could execute through the terminal commandline. So, we can wrap the legacy code to component by describing the legacy code with ADL, which includes the commandline, execution environment and related files. Moreover, these legacy codes should be the commandline programs running on the Linux or other operating system.

The goals of our work are to develop techniques and methods for turning those legacy codes into components. It includes:

- add some specific tags into the standard ADL to take into account legacy code specificities,
- define the standard API for achieving the shift from legacy codes to components,
- define the needed server and client interfaces to manipulate and control the legacy code.

In order to describe the execution environment of the legacy code and solve the portability problems, we add some specific tags in the standard ADL. Using these tags, we can know whether the target system meet the need of the legacy code, such as the Operating System, CPU or Memory. When some errors happen, we should migrate the legacy code Component to another node. The `relatedFiles` tags will be used to indicate the useful files of the legacy code, which should be transferred to remote node.

According to the GCM's ADL and the JSDL (Job Submission Description Language), we extend the ADL and add some specific tags. This is the added tags' DTD. We can also define the new tags to the ADL and extend the DTD, according to the new need of wrapping the legacy codes to components. This is an example of a legacy code ADL:

```
<definition name="legacyCode">
  <interface name="r" role="server"
        signature="net.coregrid.gcm.legacyComponent.legacyCode" />
  <content
  class="net.coregrid.gcm.legacyComponent.legacyCode.LegacyCodeClassProxy">
        <legacycode>
              <executable>/home/test/cap3</executable>
              <argument>input.seq</argument>
              <argument>-a</argument>
              <argument>20</argument>
        </legacycode>
        <resources>
              <operatingSystem type="RedHat Linux" version="9.0" />
              <CPUArchitecture type="X86" />
              <CPUTime value="30.0" />
```

```
            <CPUSpeed LowerBound="1024000000" UpperBound="2048000000" />
            <CPUCount LowerBound="1" UpperBound="2" />
            <memory LowerBound="102400" UpperBound="204800" />
            <networkBandwidth LowerBound="1024000000"
                  UpperBound="2048000000" />
            <diskSpace LowerBound="1024000000" UpperBound="2048000000" />
            <jobManager type="PBS" />
      </resources>
      <relatedFiles>
            <relatedFile name="/home/test/cap3"
                  permissions="execute,read,write"
                  deleteOnTermination="false" />
            <relatedFile name="/home/test/input.seq"
                  permissions="read,write" deleteOnTermination="true" />
      </relatedFiles>
  </content>
</definition>
```

In order to achieve the shift from legacy codes to Components and build the needed server and client interfaces to manipulate and control the code, we define some API to implement these methods. In the Figure 15, we describe the execution status transition of the legacy code in detail.

A future deliverable, D.CFI.04 Methods and Techniques for legacy Code Wrapping as Grid Components, will describe more precisely this feature.

4 Conclusion

We developed an implementation of the GCM model based on the ProActive library; this implementation is highly configurable and offers some optimizations. The architecture of the early prototype isn't definitely fixed since it was designed to be easily extended or modified. For instance, some part of the early prototype could be improved in terms of architecture and design, especially the deployment and the legacy code wrapping feature in order to allow more flexibility. In particular, the feedback from use cases will help us to bring out problems and needs.

The future GridCOMP deliverable D.CFI.06 will contain the final component framework prototype and a detailed architectural design. As well as adding some missing features in the CFI, another major point requiring improvement in the prototype design is to clearer separate the ProActive and the GCM part. One possibility will be to completely put GCM on top of ProActive; we have already begun to investigate this way. This will definitely ease the development of the ProActive/GCM implementation.

5 Bibliography

[BAU 06] FRANÇOISE BAUDE, DENIS CAROMEL, MARIO LEYTON, and ROMAIN QUILICI. "Grid File Transfer during Deployment, Execution, and Retrieval". In Proceedings of *On the Move to Meaningful Internet Systems 2006*, Montpellier, France, 2006.

[CAR 93] DENIS CAROMEL. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[DRE] Dream project. http://dream.objectweb.org

[FRAa] Fractal specification. http://fractal.objectweb.org/specification/index.html

[FRAb] Fractal ADL. http://fractal.objectweb.org/fractaladl/index.html

[GCM] D.CFI.01 - Component model presentation and specification (XML schema or DTD), *GridCOMP deliverable.*

[NFC] D.NFCF.01 - Non functional component subsystem architectural design, *GridCOMP deliverable.*

[PRO] "ProActive web site". http://proactive.objectweb.org

6 Appendix A

This is the default configuration file for the controllers and interceptors of a component in the ProActive/GCM implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentConfiguration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="component-config.xsd"
  name="defaultConfiguration">
  <!--  This is the default configuration file for the controllers and
interceptors of a component in the proactive implementation.-->
  <controllers>
        <controller>
              <interface>
  org.objectweb.proactive.core.component.controller.ProActiveBindingCon
troller
              </interface>
              <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveBindingCon
trollerImpl
              </implementation>
        </controller>
        <controller>
              <interface>
  org.objectweb.proactive.core.component.controller.ComponentParameters
Controller
              </interface>
              <implementation>
  org.objectweb.proactive.core.component.controller.ComponentParameters
ControllerImpl
              </implementation>
        </controller>
        <controller>
              <interface>
  org.objectweb.proactive.core.component.controller.ProActiveContentCon
troller
              </interface>
              <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveContentCon
trollerImpl
              </implementation>
        </controller>
        <controller>
```

```
                <interface>
  org.objectweb.proactive.core.component.controller.ProActiveLifeCycleC
ontroller
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveLifeCycleC
ontrollerImpl
                </implementation>
        </controller>
        <controller>
                <interface>
  org.objectweb.proactive.core.component.controller.ProActiveSuperContr
oller
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveSuperContr
ollerImpl
                </implementation>
        </controller>
        <controller>
                <interface>
                      org.objectweb.fractal.api.control.NameController
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.ProActiveNameContro
ller
                </implementation>
        </controller>
        <controller>
                <interface>
  org.objectweb.proactive.core.component.controller.MulticastController
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.MulticastController
Impl
                </implementation>
        </controller>
        <controller>
                <interface>
  org.objectweb.proactive.core.component.controller.GathercastControlle
r
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.GathercastControlle
rImpl
                </implementation>
        </controller>
        <controller>
                <interface>
  org.objectweb.proactive.core.component.controller.MigrationController
                </interface>
                <implementation>
  org.objectweb.proactive.core.component.controller.MigrationController
Impl
                </implementation>
        </controller>
  </controllers>
</componentConfiguration>
```