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.06 – CFI tuned prototype and final documentation (manual and detailed architectural design)

Due date of deliverable: 01 December 2008

Actual submission date: 19 January 2009

Start date of project: 1 June 2006 **Duration**: 33 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		
PP	Public	PP

Keyword List: component, GCM, grid, legacy code wrapping,
Responsible Partner: Denis Caromel, INRIA

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	DD-MM-YYYY	Template	Patricia HO-HUNE
1	14-11-2008	Draft	Cédric Dalmasso
2	27-11-2008	Draft	Bastien Sauvan
3	28-11-2008	Draft	Xiaofeng Wu
4	03-12-2008	Draft	Bastien Sauvan
5	08-12-2008	Draft	Clement Mathieu
6	09-12-2008	Draft	Bastien Sauvan
7	18-12-2008	Draft	Bastien Sauvan
8	19-01-2009	Final	Denis Caromel

Deliverable manager

- Denis Caromel, INRIA

List of Contributors

- Denis Caromel, INRIA
- Cédric Dalmasso, INRIA
- Clément Mathieu, INRIA
- Bastien Sauvan, INRIA
- Xiaofeng Wu, TU

List of Evaluators

- Marco Aldinucci, UNIPI
- Yongwei Wu, TU

Summary

This document describes the architecture of the Component Framework Implementation (CFI) prototype, which is the first implementation of the Grid Component Model (GCM) [GCM]. The CFI prototype and this document form the final deliverable D.CFI.06 of the Work Package 2 within the context of the GridCOMP research project. This deliverable is completed by two annexes:

- The sources for the legacy code wrapping as GCM components and its user documentation.
- The CFI documentation.

The CFI prototype has been implemented using the ProActive Grid Middleware [PRO] as a starting point. As a consequence, this document gives first a description of the ProActive's architecture and the model it uses. Then, a detailed description of how we used and extended this architecture to implement the CFI prototype will follow the first part.

Table of Content

1 INTRODUCTION 5

2 THE PROACTIVE MIDDLEWARE 6

2.1 ACTIVE OBJECTS MODEL 6
2.2 THE PROACTIVE LIBRARY: PRINCIPLES AND ARCHITECTURE 7
2.2.1 Implementation techniques 7
2.2.2 Semantics of communications between Active Object 8
2.2.3 Features of the library 10
2.3 CONCLUSION 10

3 ARCHITECTURE OF THE CFI IMPLEMENTATION 10

3.1 DESIGN GOALS 10
3.2 AN ARCHITECTURE BASED ON PROACTIVE'S META-OBJECT PROTOCOL 11
3.2.1 Component instance 11
3.2.1.1 Primitive components 13
3.2.1.2 Composite components 13
3.2.2 Controllers 14
MonitorController 14
PriorityController 15
3.2.3 Lifecycle 15
3.2.4 Interception mechanism 17
3.2.5 Communications 21
3.2.5.1 Optimization with short cuts ('shortcut') 21
3.2.5.2 Stream ports 23
3.2.5.3 Exporting components as Web Services 23
3.3 MECHANISM AND IMPLEMENTATION OF COLLECTIVE INTERFACES 24
3.3.1 Multicast interfaces 25
3.3.1.1 Configuration 26
Interface annotations 26
Method annotations 26
Parameter annotations 27
Available distribution policies 27
Dynamic dispatch 28
Reduction of results 28
3.3.2 Gathercast interfaces 28
3.3.2.1 Asynchronism and management of futures 30
3.3.2.2 Timeout 31
3.4 DEPLOYMENT 32
Principles 32
XML deployment descriptors 32
Retrieval of resources 34
Creation-based deployment: 34
Acquisition-based deployment: 35
Distribution of components 35
3.5 LEGACY CODE WRAPPING 35
3.5.1 Overview of the Solutions 35
3.5.2 The Framework of the Legacy Code Component 36

3.5.2.1 Characteristics of Legacy Code 36
3.5.2.2 The architecture of the Legacy Code Component 37
3.5.2.3 Description of the Legacy Code 37
3.5.2.4 Related File Operations 38
3.5.2.5 Execution Management 39
3.5.3 Features added to the GCM 40
3.5.3.1 API describing the Legacy Code 40
3.5.3.2 API for Related Files Operation 40
3.5.3.3 Resource Requirement of the Legacy Code 41
3.5.3.4 The Running Process of the Legacy Code 42
3.5.3.5 Wrap the Legacy Code to Component 42
4 CONCLUSION 42
5 BIBLIOGRAPHY 43
6 APPENDIX A 43

1 Introduction

This document is part of the deliverable D.CFI.06, which is the final outcome of the Work Package 2 of the GridCOMP project. The Work Package 2 aims at provide the reference implementation of the Grid Component Model (GCM) [GCM] defined by the CoreGRID NoE project [COR]. The GCM is an extension of the Fractal Component Model [FRAa] for the Grid. GCM components turn standard code, potentially parallel and distributed, or legacy code, into components able to be deployed and composed hierarchically. This implementation is used in Work Package 3 to implement non-functional GCM features [NFC] and is illustrated in the use cases [UC]. This deliverable is useful for anyone who wants to understand and use the Component Framework Implementation (CFI) prototype.

The whole deliverable D.CFI.06 is made of two main parts:

- This document which describes the architecture of the CFI prototype.
- A zip file, D.CFI.06_Bundle.zip, containing the sources of the last version of the CFI prototype, provided through the ProActive Grid Middleware [PRO] release 4.0.2, since this prototype is build upon ProActive.

And two annexes:

- Another zip file, D.CFI.06_LegacyCode.zip, containing both the sources for legacy code wrapping as GCM components and its user documentation. The legacy code wrapping is provided separately as this feature is independent of the CFI prototype. Nevertheless, the architecture is detailed in this document.
- A PDF file, D.CFI.06_CFI-documentation.pdf, providing the CFI documentation which therefore is not provided in this document for the reason explained below.

To ease the distribution of the document in different format (PDF and HTML) we use the DocBook[1] technology. As a consequence, we can not include the CFI documentation in this document. The documentation is available in a separate file, GridCOMP_D.CFI.06_CFI-documentation.pdf. The CFI features implemented until now are documented. The documentation contains:

- Documentation of the GCM deployment framework which implements ETSI standards.
- Technical documentation describing how to use each feature included in the CFI.
- A tutorial providing a user guide explaining how to create primitive and composite components along a simple example.

This documentation related to features developed in the frame of GridCOMP is also available in the ProActive user documentation.

[1] DocBook is an XML language for technical documentation, http://www.oasis-open.org/docbook/

As mentioned above, the present document describes the implementation architecture of the Grid Component Model. 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.

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.

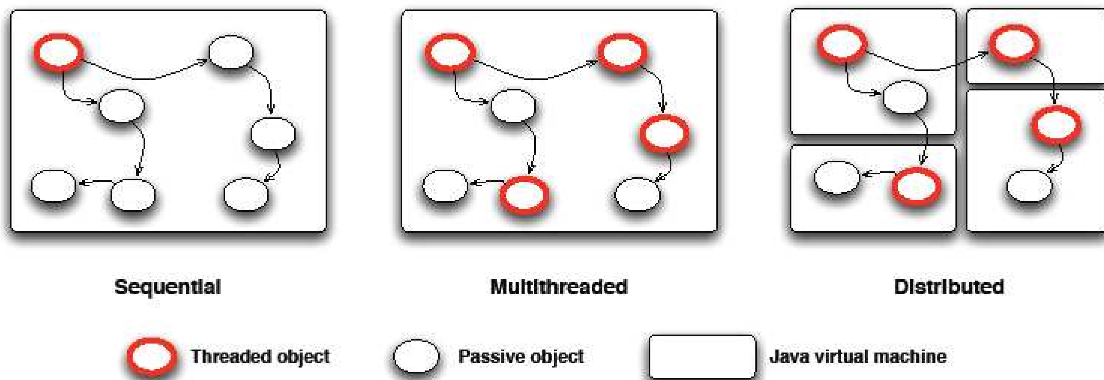

Figure 1 Seamless sequential to multithreaded to distributed objects

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.

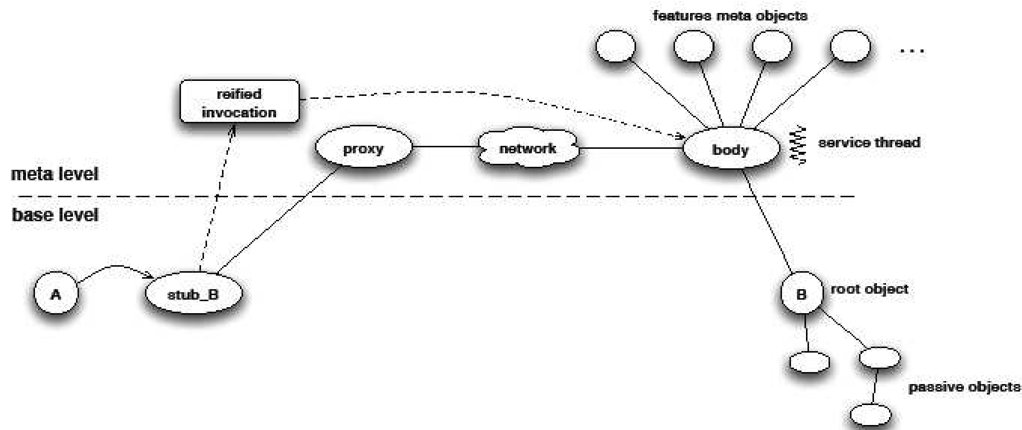

Figure 2 Meta-Object Architecture

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.

The *body* is responsible for receiving calls on the active object, storing these calls in the *queue* of pending calls (also called requests). It will execute these calls in an order specified by a specific synchronization policy. If no specific synchronization policy is provided, calls are managed in a FIFO manner (first come, first served). The *body* is not visible from the outside of the active object therefore the active object looks exactly like a standard object

from the user's perspective. It is important to note that no parallelism is provided inside an active object. This is an important decision in the design of ProActive which enables the use of pre and post conditions and class invariants.

An active object is always indirectly referenced through a *proxy* and a *stub* which is a subtype of the root object. The *proxy*'s main responsibility is to generate future objects for representing future values, transform calls into request objects (in terms of meta-object programming, this is a reification) and perform deep-copy of passive objects passed as parameters. The passive objects are not shared between subsystems. Any call on a remote active object using passive objects as arguments leads to a deep-copy of the passive objects on the subsystem of the remote active object. The role of the *stub* is to reify all the method calls that can be performed through a reference to the active object. Reifying a call simply means constructing an object (in our case, all reified calls are instance of class MethodCall) that represents the call, so that it can be manipulated as any other object. Thus, 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.

However, the use of the *stub*, *proxy*, *body*, and *queue* is transparent. ProActive manages all of them, with the user accessing the active objects in the same way as passive objects.

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. 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. By the way, since the GCM prototype is based upon the ProActive library, GCM components use implicitly this communication mechanism.

- *Synchronous invocation*:
    - the method returns 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 not detailed in this document because it is not used in the CFI. The ProActive deployment framework has several drawback and do not provides a fully and easy interoperable way to deploy application on a grid as required in the GCM definition. The main defects are the complexity to write deployment descriptors and the incapacity to easily reuse already written deployment descriptor files with another application or infrastructure. A new framework, named “GCM deployment” and detailed in the “Architecture of the CFI implementation” part, has been designed and implemented to meet those requirements. Nevertheless, the CFI is fully compatible with the ProActive deployment.

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 prototype of component framework.

3 Architecture of the CFI implementation

In this section, we describe the architecture of the CFI, which implements the GCM. We named this implementation ProActive/GCM. This 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.
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 that are achieved by trading-off between dynamicity (the possibility to dynamically reconfigure the applications, or parts of the applications) and

efficiency (direct or multithreaded invocations)An architecture based on ProActive's Meta-Object Protocol. These optimizations, such as the shortcut mechanism, are described throughout this document.

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 implemented as 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, 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 3. 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.

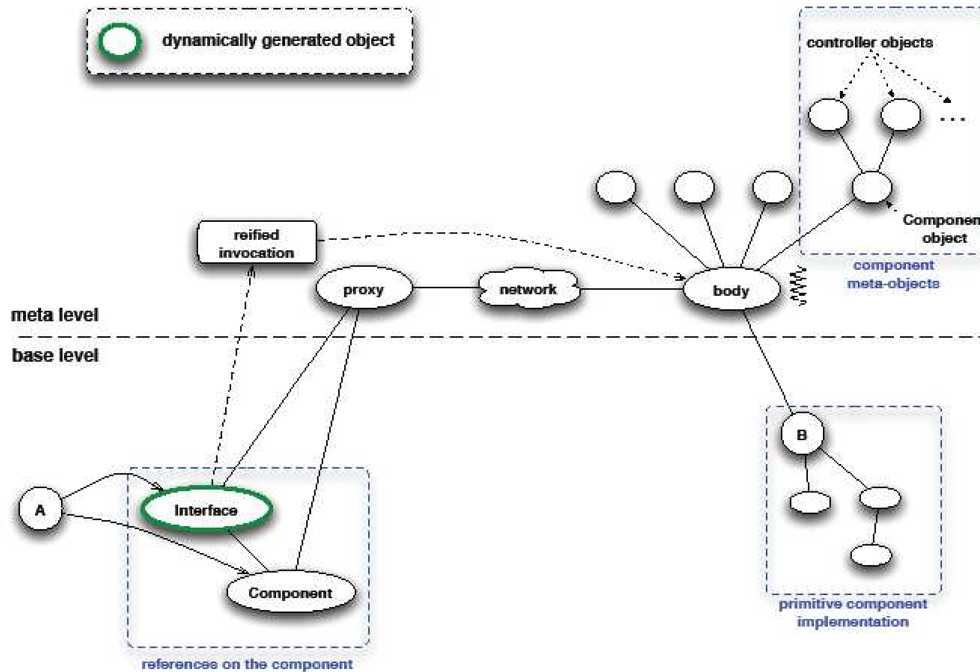

Figure 3 ProActive Meta-Object architecture for primitive components

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 3 represents an instance of a ProActive/GCM primitive component and Figure **4** 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...).

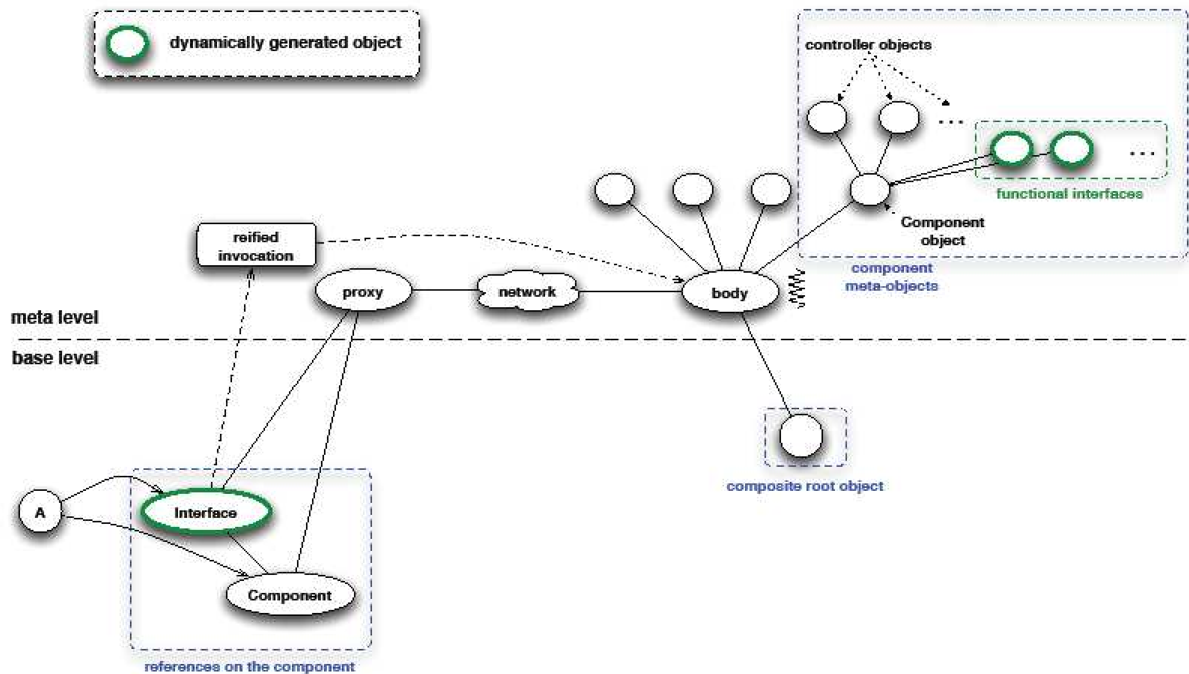

Figure 4 ProActive Meta-Object architecture for composite components

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 3. 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 **4** represents an instance of a composite component. A composite component is a structuring component which does not have any business code. There is only one exception case allowing a composite component to have an implementation class: if the composite component has an `AttributeController` and therefore, the provided class is the implementation of the `AttributeController`. 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 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. Some other controllers are related to the features offered by this implementation: migration, management of gathercast/multicast interfaces and monitoring.

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>
```

MonitorController

The org.objectweb.proactive.core.component.controller.MonitorController is an optional controller which can provide various statistics related to a given method of the server interface of a component. It is needed for GCM autonomic features and provides more information to user in the GIDE [GID] tool.

The statistics provided by the MonitorController allow users to be informed in real time on the Quality of Service (QoS) of a component and thus, eventually, to decide to reconfigure their application to improve the global performance.

In the ProActive library, each Active Objects, and so each component in the CFI, emits JMX notifications at the time of the arrival and the departure of a request in the incoming queue of a method of the Active Object, at the end of execution and when updating a future. JMX is a notification mechanism, based on java events, that allows alerts to be sent to client management applications. The MonitorController uses this feature to compute the statistics of the component. For each method of each server interface of a component, the MonitorController creates an instance of org.objectweb.proactive.core.component.controller.MethodStatistics. This instance will then stock all the generated ProActive JMX notifications related to the method and, thus, can provide the different statistics available:

- The current number of pending request in the queue.

- The average number of requests per second for the last past X milliseconds or since the beginning of the monitoring.

- The latest service time in milliseconds. In the case of composite component, this service time is related to the real execution time, i.e. to the time the subcomponent has taken to execute the request. Moreover, if the interface is an internal multicast interface, the service time retained is the one of the subcomponent which has taken the most of time to execute the request.

- The average service time in milliseconds during the last N method calls or in the last past X milliseconds or since the beginning of the monitoring.

- The latest inter-arrival time in milliseconds.

- The average inter-arrival time in milliseconds during the last N method calls or in the last past X milliseconds or since the beginning of the monitoring.

- The average permanence time in the incoming queue in milliseconds during the last N method calls or in the last past X milliseconds or since the beginning of the monitoring.

- The list of all the method calls (server interfaces) invoked by an invocation on this method.

PriorityController

In order to add the possibility of having Non Functional prioritized requests, a new controller has been implemented: `org.objectweb.proactive.core.component.controller.PriorityController`. This feature has been added to solve issues occurring for instance during reconfiguration or for autonomic features developed in the WP3. Using this controller, non functional requests may have a different priority and can pass other requests in the queue. Thus, in a first step, the request types have been extended and a priority order has been decided. Now, by using the priority controller to manage the priority of each method exposed by a component, requests can be:

- Functional requests, which always go at the end of the queue.
- Standard Non Functional requests (NF1), which also go at the end of the queue.
- Non Functional prioritized requests (NF2), which can pass the Functional requests but not pass the other Non Functional requests.
- Non Functional most prioritized requests (NF3), which can overtake all the other requests.

3.2.3 Lifecycle

A component has a lifecycle which is managed by a controller allowing to set the state of the component:

- Stopped: only control requests are served.

- Started: all requests, control and functional, are served.

This lifecycle is implemented by customizing the activity of the active objects.

In the context of components, we distinguish the component activity (the non-functional activity) from the functional activity. The component activity corresponds to the stopped state of the lifecycle of the component (i.e. only control requests are served). The functional activity is encapsulated and starts when the lifecycle is started. 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. This is illustrated in Figure 5.

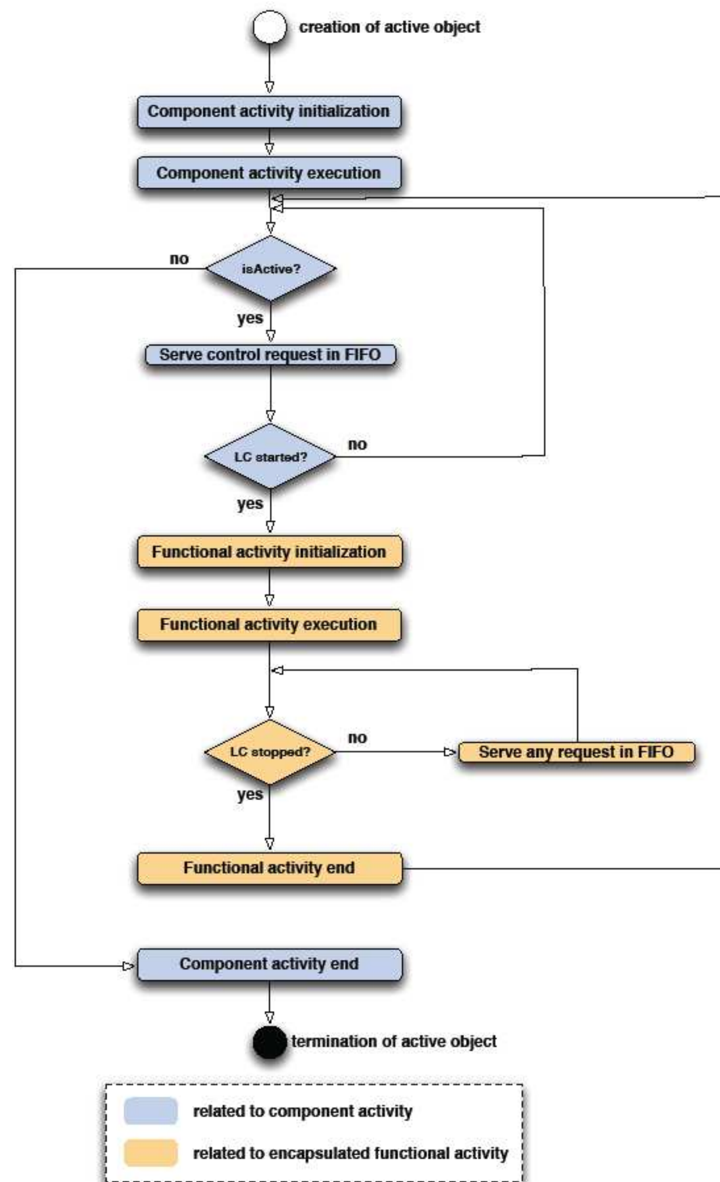

Figure 5 Default component activity

By default in ProActive, an active object is active (the `isActive()` condition is true) until the `terminate` method is called. With components, the `isActive()` condition is overridden when the component is in the functional activity and corresponds then to the state of the lifecycle. During the component activity, the `isActive()` condition reacts as for any active objet.

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.

Thus, in term of components, the component activity may be customized by implementing the `ComponentInitActive`, `ComponentRunActive` and `ComponentEndActive` java interfaces. By default, the component activity initialization and the component activity termination are done only one time:

- The initialization phase is done during the instantiation of the component (directly followed by the component activity execution). From this moment on, the component is in the active state in term of activity of active object (the `isActive`() condition on the Figure 5 Default component activity is true).
- The termination phase is done when the `isActive`() condition is false, i.e. when the `terminate` method of the active object representing the component is called.

Second, the functional activity may also 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 `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 false, i.e. when the lifecycle is stopped.

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. With these capabilities, the message may be inspected but can not be modified. This mechanism 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 3 and Figure **4**) 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 6.

```
<?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>
```

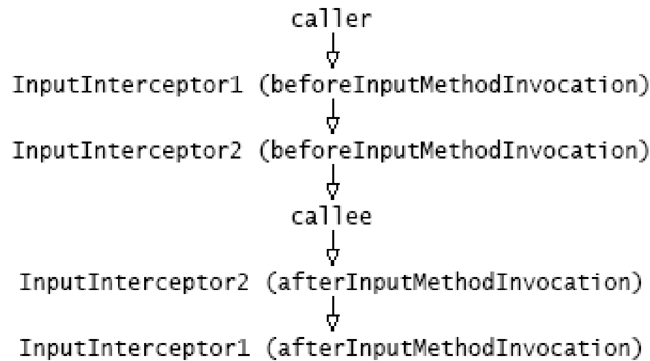

Figure 6 Execution sequence of an input interception

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 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 -->
  <!-- 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>
```

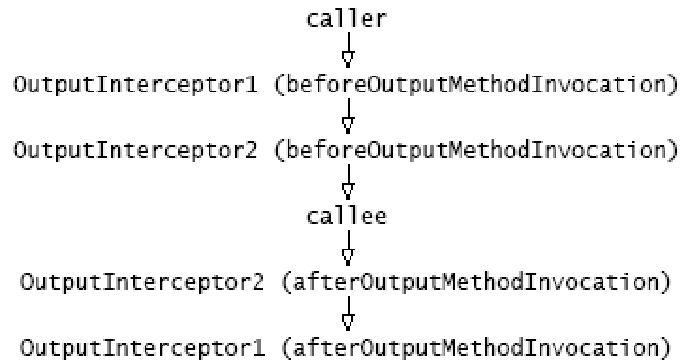

Figure 7 Execution sequence of an output interception

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, or through web services. 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.

As well as standard communication, the 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 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 event, data, 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. 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).

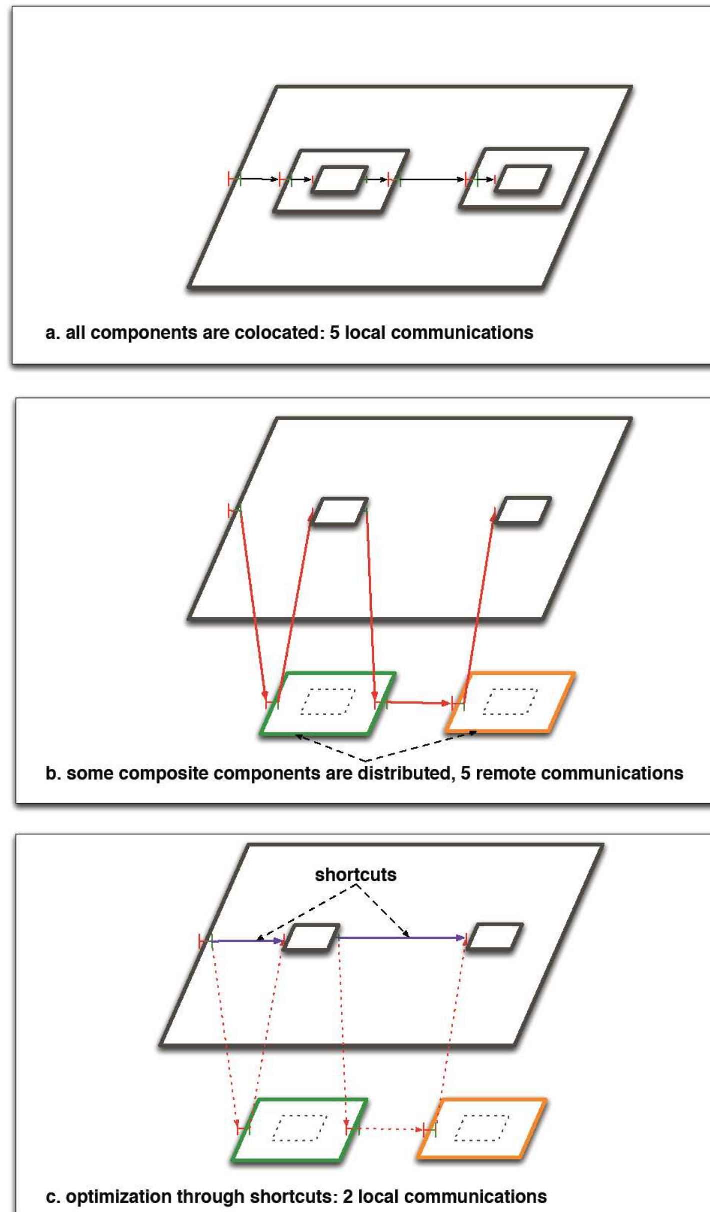

Figure 8 Using short cuts for minimizing remote communications

For instance, in Figure 8, 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.

3.2.5.2 Stream ports

A first support for stream ports is available by using the Java interface org.objectweb.proactive.extensions.webservices.StreamInterface as a tag on the java interface definition of a component interface. During instantiation of a Fractal interface type, the implementation ensures for each interface implementing the StreamInterface that all methods it defined have a void return value, otherwise the type creation failed. At the moment, there is no specific communication optimization, the provided stream interfaces just allow to express in the design the stream behaviour of a port.

3.2.5.3 Exporting components as Web Services

The ProActive middleware offers the possibility of exporting each active object as web service. Since, in our implementation, each component is implemented as an active object, component can also be easily exported as web services.

A web service is a software entity, providing one or several functionalities that can be exposed, discovered and accessed over the network. Moreover, web services technology allows heterogeneous applications to communicate and exchange data in a remotely way. In our case, the useful elements, of web services are:

- The SOAP Message: it is used to exchange XML based data over the Internet. It can be sent via HTTP and provides a serialization format for communicating over a network.

- The HTTP Server: HTTP is the standard web protocol generally used over the 80 port. Since ProActive 4.0.0, each ProActive runtime embeds a Jetty web server. This avoids users to install their own web server.

- The SOAP Engine: a SOAP Engine is the mechanism responsible of making transparent the unmarshalling of the request and the marshalling of the response. Thus, the service developer doesn't have to worry with SOAP.

- The client: Client's role is to consume a web service. It is the producer of the SOAP message. The client developer doesn't have to worry about how the service is implemented.

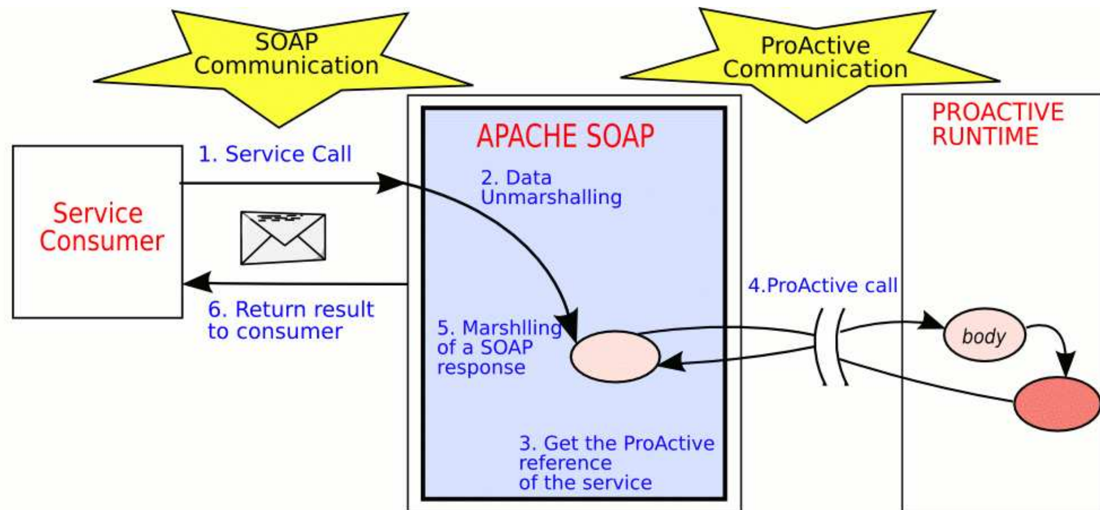

Figure 9 The figure shows the steps when an active object is called via SOAP.

However, there is a notable difference with active objects. When exposing a component as a web service, all the methods of all its client interfaces are automatically exposed as web services while with active objects, only one method can be exposed at once.

To export a component as web service, only one condition is required: the component must be started. Once the component started, the exporting is done in one call with the `org.objectweb.proactive.extensions.webservices.WebServices` API:

```
WebService.exposeComponentAsWebService(Component component, String url,
String componentName)
```

In a same manner, unexpose a web service is done with the method:

```
WebService.unExposeAsWebService(String componentName, String url,
Component component)
```

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

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.

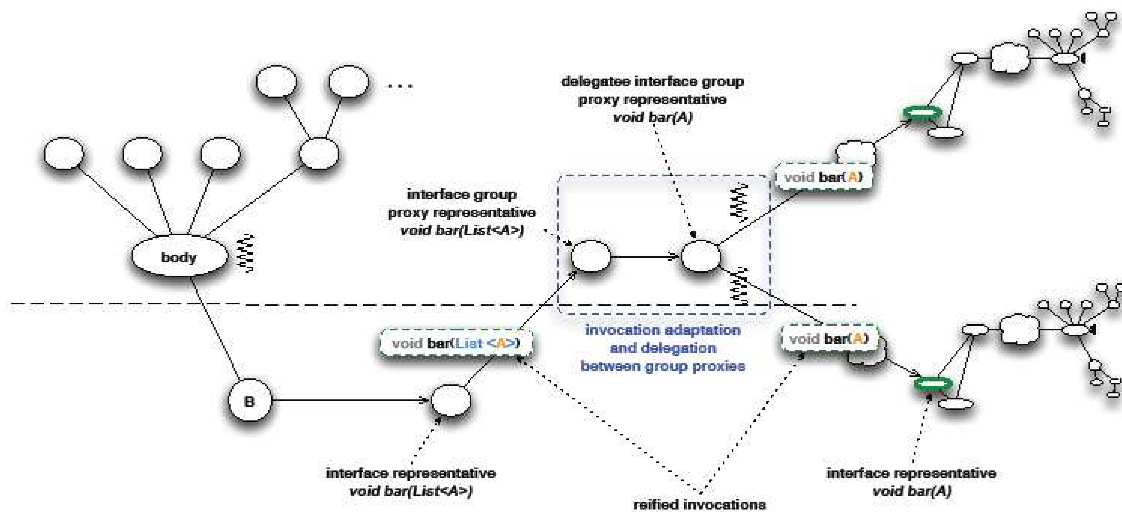

Figure 11 Adaptation and delegation mechanism for multicast invocations

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

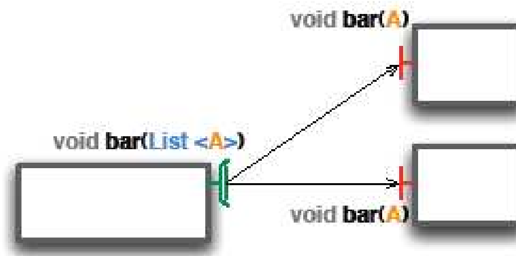

Figure 12 An example of multicast interfaces: the signature of an invoked method is exposed, and in this case exhibits a scattering behaviour for the parameters

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

Five 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.
- **UNICAST** sends one value of the list of parameters to only one of the connected server interfaces. The index of the argument to send and the server interface are specified by using a custom controller that extends MulticastController.
- **RANDOM** distributes each element of the list of values in a random manner.

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))
```

Dynamic dispatch

Moreover, as the implementation of multicast interfaces reuses the existing mechanism for typed group communications in ProActive, it benefits from features offered by this mechanism and in particular the dynamic dispatch.

First step is the partitioning of parameters according to the distribution mode as previously described. A set of tasks is generated, corresponding to the given partitioning scheme. The dispatch operation follows; it maps generated tasks to connected server interfaces, using one of the available dispatch modes: broadcast, round robin, random, custom or dynamic. With this last one, buffered tasks are statically allocated to connected server interfaces using the default allocation mode. Then, remaining tasks (un-buffered) are dynamically allocated to most appropriate connected server interfaces which increase the global performance of the execution. The buffer size can also be configured. The dispatch policy is still specified through an annotation, org.objectweb.proactive.core.group.Dispatch, at the method level:

```
@Dispatch(mode = DispatchMode.DYNAMIC, bufferdSize = myBufferSize)
```

Reduction of results

Usually, when calling a method on a multicast interface, the provided result, if there is a result, is a list of values. But, with the reduction mechanism, developer can choose to reduce the received results, i.e. gather and/or perform some operations on the list of values; for instance compute the average on a list of int and eventually return a double as result. In order to use it, the specific annotation org.objectweb.proactive.core.component.type.annotations.multicast.Reduce must be set at the method level and must specify the mode to be used. Two modes are provided in the class org.objectweb.proactive.core.component.type.annotations.multicast.ReduceMode:

- SELECT_UNIQUE_VALUE, which considers that the list contains just one value and returns this value.
- CUSTOM, which allows the developer to define its own reduction algorithm. This algorithm must be defined in a class implementing the org.objectweb.proactive.core.component.type.annotations.multicast.ReduceBehavior interface.

Thus, the reduction mechanism is used as follow:

```
@Reduce(reductionMode = ReduceMode.SELECT_UNIQUE_VALUE)
```

If the reduction fails, a org.objectweb.proactive.core.component.exceptions.ReductionException is raised.

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.

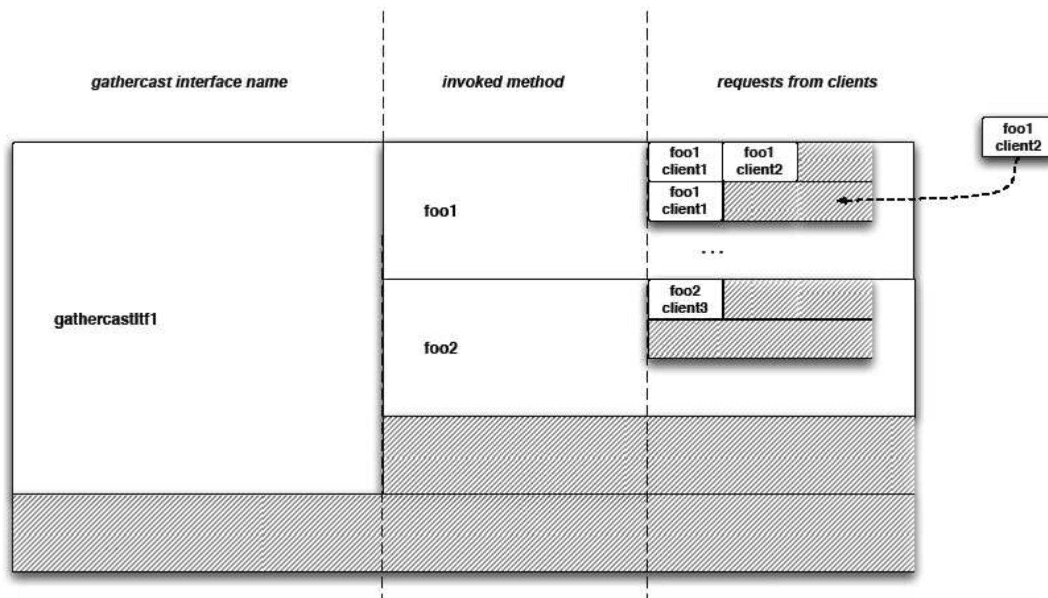

Figure 13 Data structure for the buffering of requests in gathercast interfaces

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.

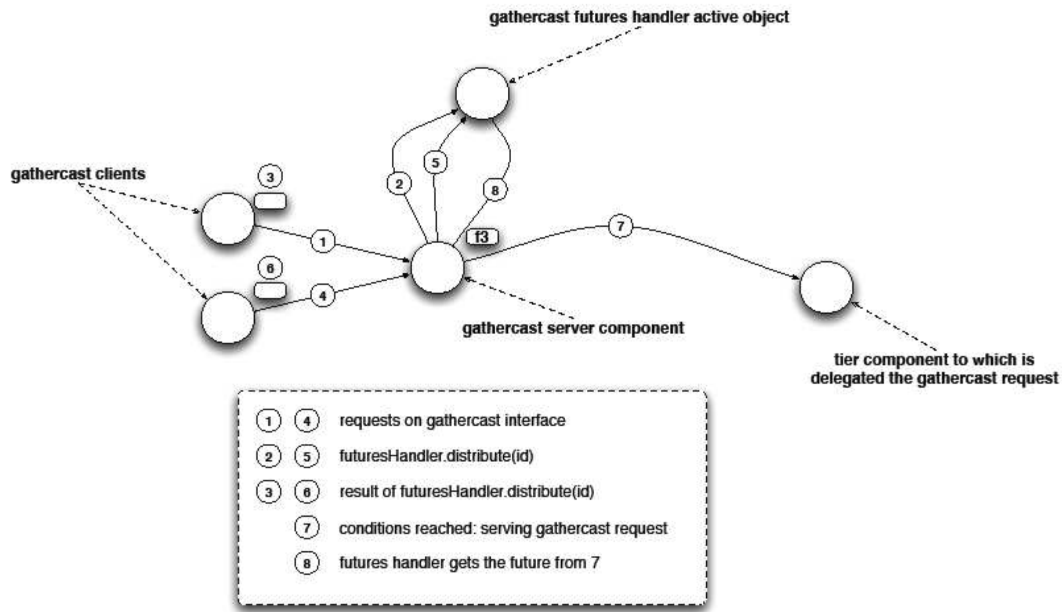

Figure 14 Management of futures for gathercast invocations

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.

Nevertheless, a waitForAll mode is available for the MethodSynchro annotation in order to relax the synchronisation constraints on gathercast interfaces. By using the waitForAll mode, developer can choose to have a gathercast interface which will create and execute an

invocation on the first request received from any of the connected client interfaces and therefore to not wait requests from other connected client interfaces. The waitForAll mode takes as parameter a boolean indicating if the method must wait or not requests from all binded client interfaces:

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

Moreover, using this mode does not require any specific changes on the client and server interfaces.

Actually, this provides to gathercast interfaces a symmetrical behaviour to the multicast unicast mode.

3.4 Deployment

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. The CFI 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 CFI documentation provided as annex to this deliverable.

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 the CFI relies on two XML descriptors. Those descriptors have been standardized by the ETSI as "GCM Deployment Descriptor" and "GCM Application Descriptor".

The GCM Deployment Descriptor defines a set of physical resources to be used by the application. It allows to describe:

- the way to create or to acquire JVMs,
- the way to register or to lookup JVMs.

The GCM Application Descriptor describes the application. It defines an application profile (ProActive, MPI, Executable etc.) and all the options associated with the given profile. It also introduces 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 VN 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.

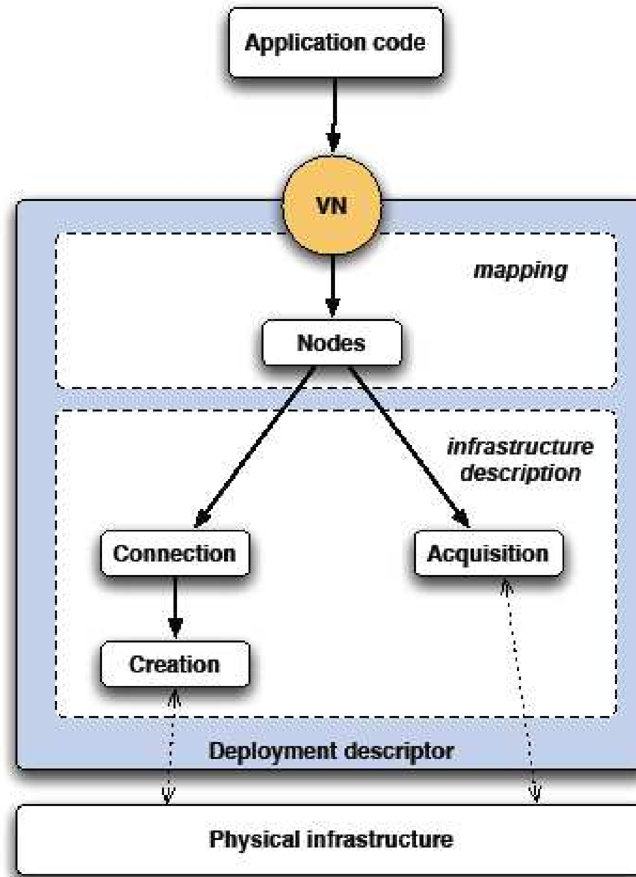

Figure 15 The deployment framework in the CFI.

Figure 15 summarizes the deployment framework provided in the CFI. 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 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, oarsh,

- lsf, pbs, sun grid engine, oar, prun,
- glite,
- Microsoft CCS,
- Amazon EC2.

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 start a ProActive runtime on each. It is also possible to start a process on the local machine

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.

Distribution of components

The deployment process is based on both the Fractal ADL [FRAb] capabilities and the 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 deployment mechanism.

3.5 Legacy code wrapping

We can deploy the legacy program as a component without re-engineering the code, or even requiring access to the source files.

In order to take into account the legacy code requirements, we should provide interfaces to specify the attributes it possesses to the component. We also design some methods to turn legacy codes into components by providing some standard APIs to manipulate and control the legacy codes.

3.5.1 Overview of the Solutions

Grid computing offers seamless integration of hardware and software resources, databases, special devices and services into a geographically distributed environment. It facilitates flexible, secure and coordinated resource sharing among participants. It has many potential advantages for solving computation intensive tasks or supporting collaborative works.

A grid computing environment requires special grid applications for standard applications to utilize the underlying grid middleware and infrastructure. Most grid projects develop new applications, or significantly re-engineered existing codes in order to make them be able to run on their platforms. However, since parallel code has been widely used in both scientific and industrial fields, deploying legacy applications on this new platform is required and practical. Unfortunately, many companies and institutions neglect or skirt this problem, and the consequence of such decisions is making lots of existing application programs unable to

run on the new platform. In order to obtain existing functionalities running on GridCOMP environment, with the least effort and cost, the legacy applications should be reused in a grid computing environment.

Our approach is based on creating a component capable of executing legacy code over grid. We want to deploy the legacy codes as components without re-engineering the code, or even access the source files.

A component, in this context, is a software module with a standardized description of what it needs and what it provides, which can be manipulated by tools for composition and deployment. It enables legacy codes written in any source language (FORTRAN, C, Java, etc.) to be easily deployed as grid components without any programming effort from the end-user. A component must be deployable on any machine flexibly, without mentioning the computing environment. Actually, we assume that acquired resources are suitable for the available legacy code binaries; this could be done using constraint at deployment time (see section 3.5.3.3). Then, when users want to execute legacy codes, they will be able to deploy the components to the remote computers immediately. A component must have complete function modules for executing the legacy code. If so, the users will be able to manipulate the components and control the execution state of the running process that belongs to the original legacy code. Therefore, this solution is much more dynamic and flexible than ad-hoc solution.

3.5.2 The Framework of the Legacy Code Component

3.5.2.1 Characteristics of Legacy Code

Legacy code is understood as a black box with specified input and output parameters plus some environmental requirements. Only the executable code is required, in this case, and there is only a user-level understanding of the application. This scenario is very common in both scientific and business applications.

The assumed general characteristics of any legacy code and its consequent wrapped solution are as follows:

1. The source code is not available.
2. The program is poorly documented and the necessary expertise to do any modifications has long left the organization.
3. The application has to be ported onto the grid within the shortest possible time and smallest effort and cost.
4. The functionalities are offered to partner organizations but the source is not.

In order to wrap the legacy codes into GCM Components and make them executable over the grid, we have researched the characteristics of the legacy code and then provided some solutions.

Executing legacy code over the grid is very important and necessary. There are many kinds of legacy codes, such as MPI programs, executable programs running on single computers or on clusters. However, almost all legacy code can be executed through the pattern of terminal command line. That means that we can control the running process of the legacy code through the command line tools. Moreover, most legacy codes are command line programs running on Linux or other operating system. Therefore, we can wrap the legacy code to a component by describing the legacy code; this description includes the command line execution environment

and related files. These elements are also important for executing the wrapped legacy code over the grid.

The purpose of this framework is to develop techniques and methods for turning legacy codes into components. According to our research, grid-enabling legacy code includes the following actions:

1. For the legacy code, provide some APIs in a standard interface to describe the legacy code attributes, such as the command line format and parameters.
2. For the related file operations, define some APIs in the interface to transfer the files and set the files' attributes.
3. For the resource requirement of the legacy code, include it in the "GCM application description".
4. For the running process of the legacy code, define the needed server and client interfaces to manipulate and control the legacy code.

3.5.2.2 The architecture of the Legacy Code Component

Figure 16 The architecture of the legacy code component demonstrates the architecture of the legacy code component. It includes two interfaces: the *AttributeController* and the server interface. Using these interfaces and java classes, we can create the Legacy Code Component. Through these interfaces, we can describe the legacy code, set these files' permissions and control the running process of the legacy code, we also provide a separate class for file transmission.. In the following chapters, we will describe each part in detail.

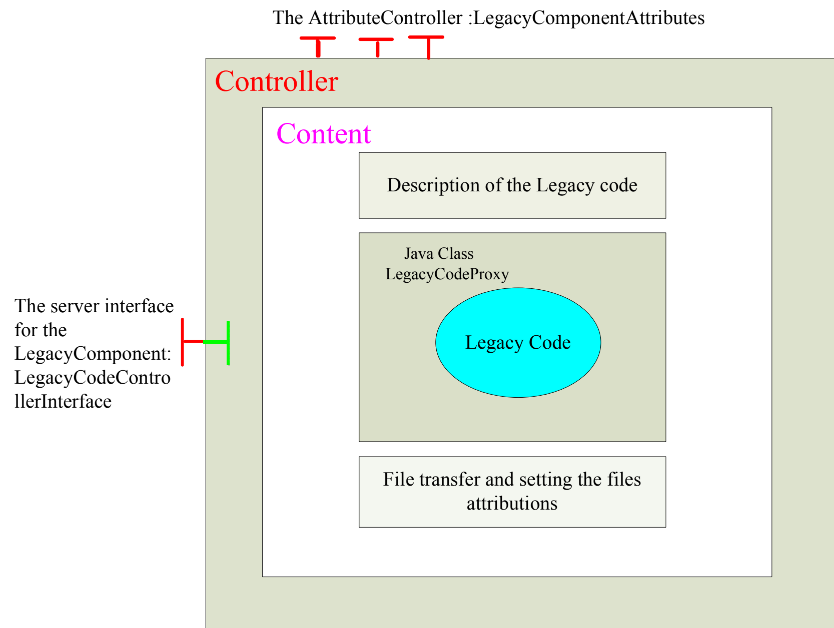

Figure 16 The architecture of the legacy code component

3.5.2.3 Description of the Legacy Code

For running the legacy code over the grid, the most important thing is to describe the legacy code, such as the command line, the execution environment, and the related files. Two methods can achieve this purpose: extending the ADL or providing some API in a standard Interface.

Extending the ADL is not a good practice if we want a flexible and extensible solution. First, when describing the specificities of the legacy code, we wanted to extend the ADL and modify the schema of the ADL. A first successful prototype was implemented which demonstrated the validity of all the identified needs for legacy code wrapping. However, we decided to refine our solution to improve and ease the wrapping for both the user and the developer. A first drawback with this solution is the difficulty to propose an ADL extension able to manage any kind of legacy code. Actually, we can be sure that additional tags would be added to the ADL, but it may also happen that some tags could be missed. Furthermore, we do not want this extension to be linked with a specific version of the ADL. In summary, extending and maintaining the ADL requires a lot of effort. Therefore, we propose a solution which is more flexible and easy to adapt.

Our solution is to provide a predefined component type for legacy code wrapping (see the user documentation of the annex D.CFI.06_LegacyCode.zip, Section 3.1: Standard legacy code wrapper component). This component has an attribute controller which defines all parameters. If you want to create a legacy code component you have to create a component extending this predefined type and set the correct attributes.

- Using the ADL, you only have to write an ADL file extending this type and set the right attributes to configure it correctly (see the user documentation of the annex D.CFI.06_LegacyCode.zip, Section 3.2: Template to be filled in by the user).

- Using the API, you have to create a component with the defined factory (LegacyComponent class), and next set attributes with the AttributeController interface (see the user documentation of the annex D.CFI.06_LegacyCode.zip, Section 1: Wrapping code example).

Through the standard interface, all things which are specific to legacy code wrapping can be managed by the component.

3.5.2.4 Related File Operations

The Related File Operations (RFO) are important for the execution of the legacy code component over the grid. They define some APIs to transfer files and set the attributes of those files.

For the file transfer, RFO implements legacy code and input files transfer between the user site and target system. In our design, we reuse the file transfer mechanism provided by the ProActive middleware. Currently supported protocols for file transfer deployment include the ProActive File Transfer Protocol (PFTP), SSH, RSH and Nordugrid. The start of the File Transfer will take place before the deployment of the component or after the successful execution of the legacy code at the target computing node.

For the file attribute settings, we should set the permissions of the related files, such as “read”, “write” or “execute”. It will ensure that the legacy code has the right permission and executes successfully. By the end of the running process of the legacy code, we should delete all the files no-longer needed. Depending on the attribute settings of those files, the component could do the deletion automatically.

It should be noticed that some interesting things should be achieved, such as transferring the ProActive libraries into the deploying machine using an on-the-fly style. This would enable

the deployment of the components on remote machines without having ProActive pre-installed. Even further, when the network allows it, it would also be possible to transfer other required libraries like the JRE (Java Runtime Environment) to the target system.

There is one protocol, which behaves differently from the others mentioned above, the ProActive File Transfer Protocol (PFTP). The main advantage of using PFTP is that no external copy protocols are required to transfer files during deployment. Therefore, if the grid infrastructure does not provide a way to transfer files, a file transfer deployment can still take place by using the PFTP. On the other hand, the main drawback of using PFTP is that ProActive must already be installed on the remote machines, and thus on-the-fly deployment is not possible.

3.5.2.5 Execution Management

In order to run the legacy code successfully and control the running process of the legacy code component, we define the execution management module, which contains the execution status of the legacy code, defines the transition between the different execution status, and the standard API for controlling the program.

When wrapping the legacy code to the component and running it, the legacy code includes status states such as UNSTARTED, RUNNING, KILLED and FINISHED. Each status defines the current state of the legacy code. It helps to keep the consistency of the legacy code and a better control of the code when multiple instances are running. We also define a standard API to control the component execution, with methods such as `startLegacyCode()`, `killLegacyCode()`, `restartLegacyCode()`, `getStatus()`. According to this API, the user could control the running process of the legacy code and have a good interactive interface. Meanwhile, the component monitors the execution state and sends feedback to the user. Figure 17 shows the transition between the different execution status of the wrapped legacy application by using the mechanisms provided by the execution management module.

After running the legacy code successfully, the result files could be obtained and transferred to the users by using a method from the standard API.

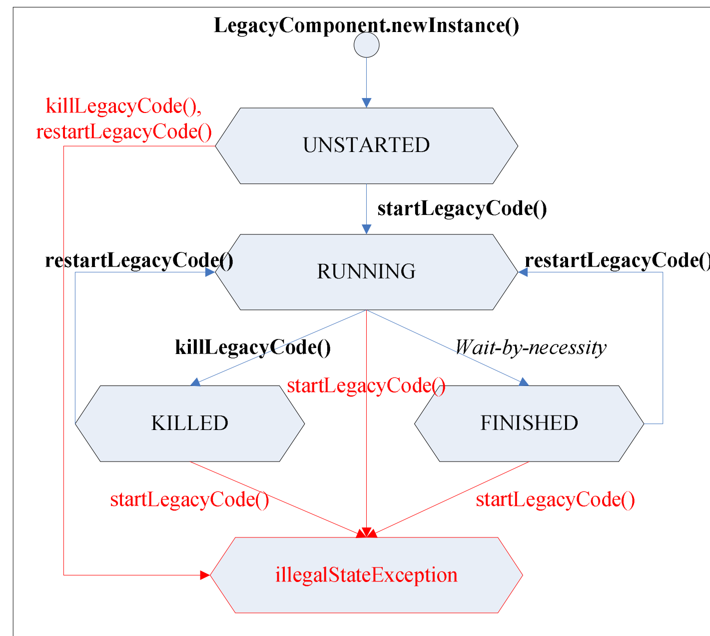

Figure 17 The execution status transition of the legacy code

3.5.3 Features added to the GCM

3.5.3.1 API describing the Legacy Code

There should be some methods in the interface to describe the legacy code, such as the command and its parameters.

```
public interface LegacyComponentAttributes extends AttributeController {
        public void setComment (String value);
        public String getComment ();

    public void setExecutable (String value);
        public String getExecutable ();

    public void setParameters (String value);
        public String getParameters ();

    public void setCommandLine(String CommandLine);
        public String getCommandLine();

            public void setFilePermission (String permission);
      public void setFileDelete (String delete);

}
```

3.5.3.2 API for Related Files Operation

Part of the API is used to transfer the related file and retrieve the result file.

```
package org.tsinghua.gcm.legacyComponent.relatedfile;
```

```
public class FileTransfer {
  //The push methods transfer a file/directory or files available on the local
  //node(srcNode) to the specified remote node(dstNode)
          public static void push (File [] srcFiles, Node dstNode, File [] dstFiles);
          public static void push (Node srcNode, File [] srcFiles, Node dstNode,
                          File [] dstFiles);
      public static void push(Node srcNode, File srcFile, Node dstNode, File dstFile);
      public static void push(Node srcNode, File[] srcFiles, Node dstNode, File[]
                     dstFiles);

  //The pull methods retrieve a file/directory or files located on a remote
  //machine(srcNode) to the local machine(dstNode)
          public static void pull (Node srcNode, File srcFile, File dstFile);
          public static void pull (Node srcNode, File [] srcFile, File [] dstFile);
      public static void pull(Node srcNode, File srcFile, Node dstNode, File dstFile);
      public static void pull(Node srcNode, File[] srcFiles, Node dstNode, File[]
                     dstFiles);
  }
```

3.5.3.3 Resource Requirement of the Legacy Code

For this part, we should extend the “GCM application descriptor” definition. This file coupled to “GCM deployment descriptor” files allows users to describe how to acquire resources from a given grid and deploy an application using these resources. Both are being standardized in ETSI [DIS]; therefore, the proposed solution is a working solution but not a final one. The main idea is to extend or reuse the definition of a virtual node in a “GCM application descriptor” file with the specification of the following elements:

- *operatingSystem:* specify the required operating system, including a specific version
- *CPUArchitecture:* specify the required CPU architecture,
- *CPUSpeed:* specify the required CPU speed (with lower and upper bound)
- *CPUCount:* specify the required CPU count (with lower and upper bound)
- *memory:* specify the required amount of memory (with lower and upper bound)\
- *networkBandwidth:* specify the required network bandwidth (with lower and upper bound)
- *diskSpace:* specify the required space disk (with lower and upper bound)

In fact, all these elements are often needed to know how and where to execute the legacy application; but legacy applications are not the only case where a user may need to select the node where he wants to deploy such component. Consequently, we have decided against a specific solution for the legacy code wrapping with just those requirements and use instead the work done at standardization level.

3.5.3.4 The Running Process of the Legacy Code

```
package org.tsinghua.gcm.legacyComponent.legacyCode
        //define the interface to manipulate and control the code
    public interface LegacyCodeControllerInterface {
                public LegacyCodeResult startLegacyCode();
                public LegacyCodeResult reStartLegacyCode();
                public boolean killLegacyCode();
                public String getStatus();
                public void setLegacyCodeCommand (String arguments);
    }
```

3.5.3.5 Wrap the Legacy Code to Component

```
package org.tsinghua.gcm.legacyComponent
    //wrap the legacy code to Component
    public class LegacyComponent{
                public Component LegacyComponent();
    }
```

In conclusion, the presented solution for legacy application wrapping is to create a predefined and already implemented component wrapper type to be used when executing it over the grid. The designed architecture of the Legacy Code Component Wrapping provides some interfaces to set the attributes of the legacy code and a standard API to manipulate and control the legacy code execution. This wrapping component can be easily deployed remotely to interact with other components.

4 Conclusion

This document demonstrates that we have developed an implementation of the GCM model, which is based on the ProActive library and provides all the main features of the model, such as primitive and composite components, single and collective bindings, ADL and deployment. Thus, the CFI prototype can be used to design and implement grid component based applications.

Among the future works, we will improve the possibility to have non functional components, i.e. components managing non functional aspects put in the membrane. An early version of this feature is already provided in the CFI prototype and documented in the CFI documentation. However, we did not detail the architecture in this document but we have just mentioned the feature as an on-going work since it is still under development and therefore the architecture may evolve. This feature allows developer to create controllers of a component as component themselves. Using non functional components, developer takes advantage of the structure, the hierarchy and the encapsulation provided by a component-oriented approach.

One of the main objectives of the GCM was to ensure the interoperability. The GCM deployment standard already satisfies this point at deployment time with the support of various middleware and schedulers. In addition, services offered by a GCM component can be accessed through Web Service (WS). Next steps will be support of WS bindings allowing a given GCM component to access another GCM component or application using WS. Such features will improve the interoperability at the component communication level with other middleware. A more general objective is to provide an SCA implementation with dynamicity

at runtime thanks to the GCM features. Therefore, GCM components are the building blocks for integrated SOA towards SLA and QoS.

In addition, projects such as SOA4ALL (EU) [SOA], INRIA ADT Galaxy [GAL], Pole de Compétitivité AGOS (with HP, Oracle) [AGO], and QosCosGrid (EU) [QOS] use ProActive the GCM reference implementation.

5 Bibliography

[AGO] Pole de Compétitivité AGOS, http://ralyx.inria.fr/2007/Raweb/oasis/uid91.html

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

[COR] CoreGRID Network of Excellence, European funded project, http://www.coregrid.net/

[DIS] D.DIS.02 - Standardization Strategy, *GridCOMP deliverable*

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

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

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

[GAL] INRIA ADT Galaxy, http://galaxy.gforge.inria.fr/Main/HomePage

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

[GID] D.GIDE.04 - Grid IDE tuned prototype and final documentation (manual and detailed architectural design), *GridCOMP deliverable*

[NFC] D.NFCF.05 - NFCF tuned prototype and final documentation (manual and detailed architectural design), *GridCOMP deliverable*

[PRO] “ProActive web site”, http://proactive.objectweb.org

[QOS] QosCosGrid, European project, http://www.qoscosgrid.eu

[SOA] SOA4ALL, European project, http://www.soa4all.eu

[UC] D.UC.05 - Use cases: tuned prototypes and final documentation (manual and detailed architectural design), *GridCOMP deliverable*

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.ProActiveB
indingController</interface>

<implementation>org.objectweb.proactive.core.component.controller.ProAc
tiveBindingControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.ProActiveC
ontentController</interface>

<implementation>org.objectweb.proactive.core.component.controller.ProAc
tiveContentControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.ProActiveL
ifeCycleController</interface>

<implementation>org.objectweb.proactive.core.component.controller.ProAc
tiveLifeCycleControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.ProActiveS
uperController</interface>

<implementation>org.objectweb.proactive.core.component.controller.ProAc
tiveSuperControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.fractal.api.control.NameController</interface>

<implementation>org.objectweb.proactive.core.component.controller.ProAc
tiveNameController</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.MulticastC
ontroller</interface>

<implementation>org.objectweb.proactive.core.component.controller.Multi
castControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.Gathercast
Controller</interface>

<implementation>org.objectweb.proactive.core.component.controller.Gathe
rcastControllerImpl</implementation>
        </controller>
```

```
        <controller>

<interface>org.objectweb.proactive.core.component.controller.MigrationC
ontroller</interface>

<implementation>org.objectweb.proactive.core.component.controller.Migra
tionControllerImpl</implementation>
        </controller>
        <controller>

<interface>org.objectweb.proactive.core.component.controller.MonitorCon
troller</interface>

<implementation>org.objectweb.proactive.core.component.controller.Monit
orControllerImpl</implementation>
        </controller>
    </controllers>

</componentConfiguration>
```