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)

**ANNEX 1: CFI documentation**

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

# Part I. Programming With Components

## Table of Contents

# Chapter 1. Introduction

## 1.1. Overview

Computing Grids and Peer-to-Peer networks are inherently heterogeneous and distributed, and for this reason they present new technological challenges: complexity in the design of applications, complexity of deployment, reusability, and performance issues.

The objective of this work is to provide an answer to these problems through the implementation for ProActive of an extensible, dynamical and hierarchical component model, Grid Component Model (GCM) based on Fractal [http://fractal.objectweb.org]. The GCM was defined by the CoreGRID NoE project [http://www.coregrid.net/] and is available here [http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf].

This part documents the ProActive/GCM reference implementation developed by the GridCOMP European project [http://gridcomp.ercim.org/].

## 1.2. Programming with components: the Fractal component model

Fractal defines a general conceptual model, along with a programming application interface (API) in Java. According to the official documentation, the Fractal component model is '**a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces**'.

Fractal is a component model. A component is a software module offering predefined services, and able to communicate with other components. The Fractal component model is hierarchical, so components can be either primitives or composites. A composite can contain one or many inner components (primitive or composite). Each component may define what it needs and provides with its client and server interfaces. Furthermore, server interfaces may be functional interfaces or a non-functional interfaces (also called controllers). Controllers are useful to manage the component. For instance, the LifeCycleController allows to control the life cycle of the component and provides methods to start or stop the component.

Here is a basic example of a system of Fractal components:

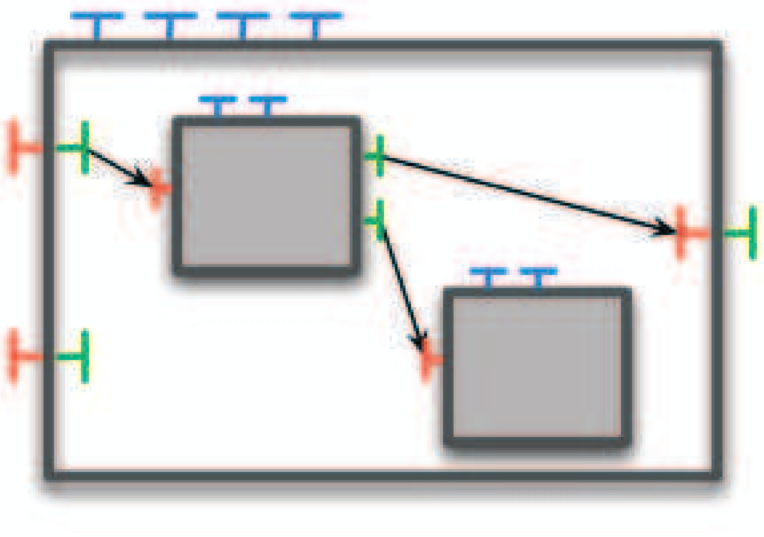


**Figure 1.1. A system of Fractal components**

In addition to that, Fractal defines an Architecture Description Language (ADL). The ADL uses an XML syntax and is a way to describe a component based system without having to worry about the implementation code.

The Fractal specification defines conformance levels for implementations of the API (section 7.1. of the Fractal 2 specification).

For a complete description of the Fractal component model, please, refer to the Fractal specification, available at http://fractal.objectweb.org/specification/fractal-specification.pdf

## 1.3. Presentation of ProActive/GCM

The Grid Component Model (GCM) defines a component model suitable to support the development of efficient grid applications. It implements the "invisible grid" concept: abstract away grid related implementation details (hardware, OS, authorization and security, load, failure, etc.) that usually require high programming efforts to be dealt with. Our implementation of the GCM is based on the ProActive library: components in this framework are implemented as active objects, and as a consequence benefit from the properties of the active object model. We named this implementation ProActive/GCM.

Thus, the previous standard system of Fractal components becomes when distributed with ProActive/GCM:

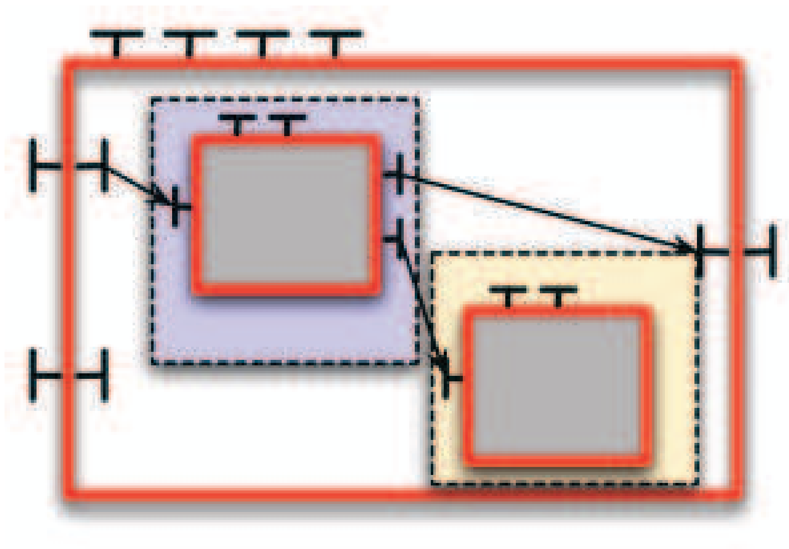


**Figure 1.2. A system of distributed ProActive/GCM components (blue, yellow and white represent distinct locations)**

The GCM is an extension of the Fractal specification, and it introduces the new features using a Fractal compliant terminology. The main features that have been developed to implements the GCM are:

- The deployment: several components in an assembly can be distributed on different nodes on several computers using transparent remote communication.
- The collective interfaces: component systems designers are able to specify parallelism, synchronization and data distribution. Collective communications refer to multipoint interactions between software entities. Collective interfaces have two types of cardinalities, multicast and gathercast.

ProActive/GCM is conformant up to level 3.2. In other words, it is fully compliant with the API. Generic factories (template components) are provided as ADL templates. We are currently implementing a set of predefined standard conformance tests for the Fractal specification.

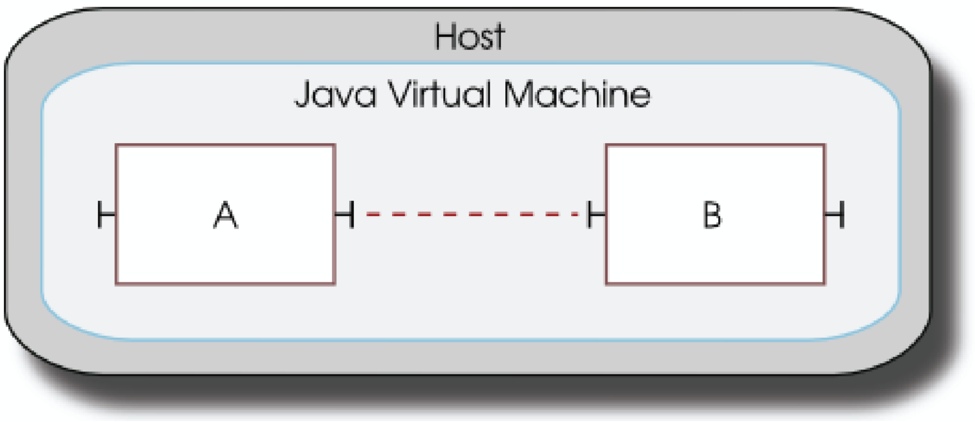
To sum it up, ProActive/GCM mainly provides:

- Creation/usage of primitive and composite components
- Client, server and non-functional interfaces (single and collection cardinalities)
- ADL support
- A deployment framework

## 1.4. GCM Basics

For starters, here is a very basic example demonstrating the separation between the code and the deployment of an application and also showing the simplicity with which the deployment can be modified.

As shown in the diagram below, in the first step, The application is just composed of two primitive components distributed into a single Java Virtual Machine.



Now, in order to use two separate Java Virtual Machine, in the deployment descriptor file, the line:

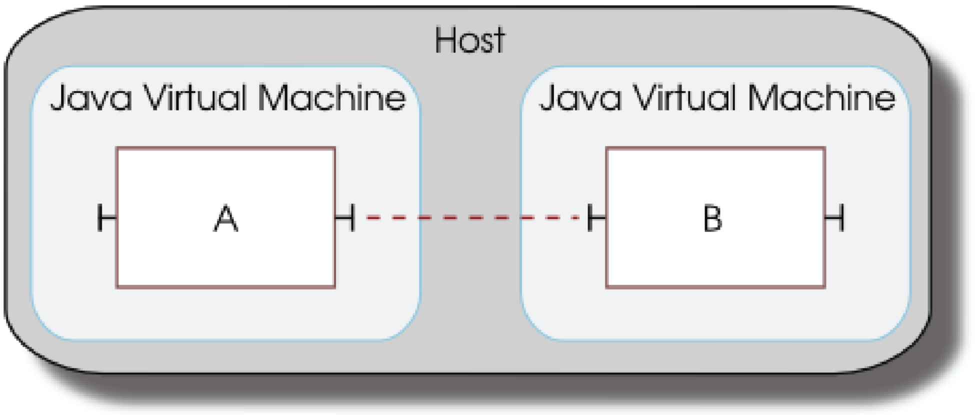
```
<host id="localhost" os="unix" hostCapacity="1"
vmCapacity="2">
```

is changed to:

```
<host id="localhost" os="unix" hostCapacity="2"
vmCapacity="1">
```

Before changing the line, the deployment descriptor indicates that there will be 1 Java Virtual Machine with 2 nodes inside the JVM.

Then, once the change made, the deployment descriptor specifies that there will be 2 Java Virtual Machines with 1 node per JVM:



All the source files are available at the end of the part in Chapter 8, *Annex*.

# Chapter 2. ProActive Grid Component Model Deployment

## 2.1. Introduction

The GCM Deployment is split in two parts, one for grid administrators and the other for grid application developers. On the grid administration side, the administrator will write a Deployment Descriptor that will describe what resources the grid provides, and how these resources are acquired. On the application side, the developer will write an Application Descriptor that will describe how the application is launched, and what resources it needs. The link between the two sides is made through references from the Application Descriptor to one or several Deployment Descriptors.

## 2.2. ProActive Deployment API

There are several ways the grid resources can be used by a deployed application. The application may require a fixed set of resources, or it may be flexible enough to work on any amount of resources, or finally may require a minimum amount of resources and yet be able to scale as more resources become available.

In all cases, the application must start by creating a GCMApplication object through PAGCMDeployment.loadApplicationDescriptor(), and call GCMApplication.startDeployment(). The application must quit through GCMApplication.kill().

### 2.2.1. Resources fixed by the application (SPMD)

In this case the application knows the amount of resources it requires. The acquisition of these resources by the application is done as follows :

- get the required virtual nodes through GCMApplication.getVirtualNode(String vnName), or GCMApplication.getVirtualNodes()
- For each virtual node, use GCMVirtualNode.getNewNodes() as many times as needed, until the virtual node has the expected numbers of physical nodes to run on. getNewNodes() will return the list of Nodes that have been acquired since the last time it was called. Calls to it should be separated by calls to Thread.sleep().

### 2.2.2. Resources fixed by the application deployer

In this case the application has no specific requirement on the resources it uses : the more the better. This is the simplest of the cases : the application only has to call GCMApplication.waitReady(). This will block until all Virtual Nodes have their configured number of physical Nodes. Note that this may block forever if a Virtual Node does not to have a limited number of nodes after which it is in 'ready' state (the Virtual Node is said to be 'greedy', GCMVirtualNode.isGreedy() will return true).

### 2.2.3. On demand Scalability

In the case the application is able to expand on new resources as they become available. This is a extension of the two other cases, in that it can work whether the application has fixed minimum requirements or not. Once the initial deployment phase is finished, the application should call GCMApplication.getVirtualNodes() to obtain the list of configured virtual nodes, and then subscribe to the node attachment notifications for each of them (GCMVirtualNode.subscribeNodeAttachment() ). In the notification handler, the application should deal with the newly acquired node appropriately.

## 2.3. GCM Deployment Descriptors

### 2.3.1. Host Information

The HostInfo data structure describes a single machine and the environment it provides, with the following information:

- userName : (string) the name of the user under which this host can be accessed
- homeDirectory : (absolute path) the home directory of the user
- os : (one of "unix" or "windows") the operating system the host is running

- hostCapacity : (positive integer) the number of processes (VM or other executable) that this host can handle (default value is 1)
- vmCapacity : (positive integer) the number of nodes a single VM on this host can handle (default value is 1)
- id : (ID) an ID identifying the host

### 2.3.2. Bridges

A bridge is meant to represent a frontend to a computing resource. Many grid architectures have such a feature : each physical machine is not accessible directly, the user must instead go through a single machine called a front-end. In a deployment descriptor, a bridge is a gateway toward either :

- a host
- a set of groups
- another bridge

A bridge is defined as a base structure meant to be derived. The base structure only defines an id (string).

#### 2.3.2.1. RSH

An RSH bridge element can have the following attributes :

- id (string) : the id of the bridge connector corresponding to this definition
- hostname (string) : the network hostname of the physical machine which acts as the bridge
- username (string, optional) : the user name under which the machine can be accessed
- commandPath (string, optional) : the path of rsh client to use

#### 2.3.2.2. SSH

An SSH bridge element can have the follow child element :

- privateKey (path string) : the file of the private SSH key needed to access the bridge

An SSH bridge element can have the following attributes :

- id (string) : the id of the bridge connector corresponding to this definition
- hostname (string) : the network hostname of the physical machine which acts as the bridge
- username (string, optional) : the user name under which the machine can be accessed
- commandPath (string) : the path of the ssh client to use
- commandOptions (string) : options to pass to the ssh command

### 2.3.3. Groups

A Group is a data structure defining a set of machines with identical configuration (like a cluster). It is meant as a base structure which can be derived in an Object-Oriented manner to implement any kind of group. There currently are two kinds of groups :

1. "direct" groups
2. job schedulers

It is therefore possible to define a standard-compliant deployment descriptor even on a grid which has its own job scheduler.

All group protocols have the following child elements :

- environment (environment) : the environment for the command and the following attributes

and the following attributes

- id (ID) : the id of the group this element represents
- commandPath (path string) : path of the command which is used to submit a job to the group protocol

#### 2.3.3.1. CCS

This group handles Microsoft's Compute Cluster Server. The CCS group definition has the following child elements :

- resources : the resources that will be allowed to the job. This element can have two children :

- cpus (positive integer) : the number of CPUs allocated for the job
  - runtime (time) : the maximum runtime allowed for the job
- stdout (path string) : path of the file where the standard output of the job will be stored
- stderr (path string) : path of the file where the standard error of the job will be stored

### 2.3.3.2. LSF

Group definition for the LSF scheduler. The LSF group definition has the following child elements :

- resource (string) : this element has the following attributes :
  - (positive integer) : number of processors requested
  - walltime (time) : maximum time allowed for the job
- processorsNumber (positive integer) : minimum number of processors requested to run the job
- resourceRequirement (string) : a resource requirement string as defined by the lsf documentation (‘lsfintro’ manpage)

It also has the following attributes

- interactive (boolean) : whether the job is interactive or not
- jobName (string) : name of the job
- queue (string) : name of the queue the job will be submitted in

### 2.3.3.3. OAR

Group definition for the OAR job scheduler [15]. The OAR group definition has the following child elements :

- resource (string) : this element has the following attributes :
  - nodes (positive integer) : number of nodes requested
  - cpu (positive integer) : number of CPUs requested
  - core (positive integer) : number of cores requested
  - walltime (time) : maximum time allowed for the job

  It also can have a string content which is passed verbatim–o the ‘--resource’ option of the oarsub command.
- directory (path string) : the working directory of the job script
- stdout (path string) : path of the file where the standard output of the job will be stored
- stderr (path string) : path of the file where the standard error of the job will be stored

It has the following attributes :

- interactive (boolean) : start an interactive job. If true, open a login shell on the first node instead of running a script (default is false).
- queue (string) : name of the queue to submit the job to.
- type (‘deploy’, ‘besteffort’, ‘cosystem’, ‘checkpoint’, ‘timesharing’) : job type – the default is ‘deploy’.

### 2.3.3.4. PBS

Group definition for the PBS/Torque job scheduler. The PBS/Torque group definition has the following child elements :

- resource (string) : this element has the following attributes :
  - nodes (positive integer) : number of nodes requested
  - ppn (positive integer) : number of CPUs requested
  - walltime (time) : maximum time allowed for the job
- nodes (positive integer) : number of nodes requested
- processorsPerNode (positive integer) : number of processors per node requested
- mailWhen (combination of Abort, Begin, End separated by ‘|’) : when to send an email (Abort : if the job is aborted, Begin : when the job is started, End : when the job terminates)
- mailTo (comma-seperated list of email addresses) : where the job status emails should be sent
- joinOutput (boolean) : if true, join the output of stderr to stdout
- stdout (path string) : path of the file where the standard output of the job will be stored

- stderr (path string) : path of the file where the standard error of the job will be stored

It has the following attributes :

- queue (string) : destination queue for the job. The argument can be of the following format :
  - queue : a queue on the default server
  - @server : the default queue on the server
  - queue@server : the queue on the given server
- jobName (string 15 chars long, no whitespace, first char must be alphabetic) : the name of the job
- interactive (boolean) : whether the job is interactive or not

### 2.3.3.5. Prun

Group definition for the PRUN run server. The PRUN group definition has the following child elements :

- resource (string) : this element has the following attributes :
  - nodes (positive integer) : number of nodes requested
  - ppn (positive integer) : number of CPUs requested
  - walltime (time) : maximum time allowed for the job
- stdout (path string) : name of the file in which the results will be printed

It has no attribute.

### 2.3.3.6. Host List

A host list can be used with SSH and RSH groups as a shorthand to specify several machine names in a compact form. The format of a host list is a whitespace-separated list of name patterns or hostnames. A name pattern describes a set of hostnames with a common root. The format is as follows.

```
<root name><interval>
```

with root name being an alphanumeric string (only letters and digits, no spaces or punctuation signs), and interval defining a set of numerical values in the form of an interval or list of values, possibly followed by an exclusion interval or list of values. The general form of an interval is:

```
[<value set>]^[<value set>]
```

or simply

```
[<value set>]
```

if no exclusion interval is needed.

A value set is a coma-seperated list of integers or integers pairs separated by a dash, meaning an interval of values. The values of an interval must be specified in increasing order, and the generated values will be in increasing order. Also, the first integer of an interval can have leading zeroes to indicate the number of digits (numbers will be padded with zeroes if needed). Some examples:

- host[0-5]: host0, host1 … host5;
- host[0-5]^[4]: host0, host1, host2, host3, host5;
- host[0-10]^[4-6]: host0, host1, host2, host3, host7, host8, host9, host10;
- host[00-5]: host00, host01, host02… host05;
- host[1, 004-7, 09]: host1, host004, host005, host006, host007, host09.

#### 2.3.3.7. RSH

The RSH Group has the following child elements :

- (host list) : the list of hosts to connect to

#### 2.3.3.8. SSH

The SSH Group has the following child elements :

- (host list) : the list of hosts to connect to
- privateKey (path string) : the file of the private SSH key needed to access the host
- commandOptions (string) : the list of options which will be passed to the ssh command

## 2.4. GCM Application descriptor

### 2.4.1. Executable

This type of application describes the launch of a stand-alone executable on the grid. It can have the following child elements :

- nodeProvider (empty element with a single 'refId' attribute) : the id of a node provider (defined in the <resources> part). There can be any number of such element.
- command : the command which will be run on the portion of the grid defined by the specified node providers. The contents of this element are described below.

This element can have the following attribute :

- instances (one of "onePerHost", "onePerVM", "onePerCapacity") : the number of instances of the command which will be run

The <command> element can have the following children (in this specified order) :

- path (path string) : the path of the executable
- arg (string) : the arg string which will be passed to the command. There can be any number of such element.
- filetransfer (file transfer) : the files which which should be transferred prior to running the command.

It can have the following attribute :

- name (string) : name of the executable. If a <path> child element is present, the value of this attribute will be appended to the value of the %lt;path> child element.

### 2.4.2. ProActive

This element describes a ProActive-based applicatin. It can have the following children :

- configuration : various configuration parameters - this element is described below
- technicalServices (technical services) : the set of technical services global to this instance of ProActive

the configuration element can have the following child elements :

- bootClasspath (simple classpath) : the boot classpath for the JVM
- java (path string) : the path to the Java executable
- jvmarg (string) : arguments passed to the JVM
- applicationClasspath (classpath) : classpath for the application
- proactiveClasspath (classpath) : classpath used to override the standard ProActive classpath computed from its installation location

- securityPolicy (relative path) : path to the Java security policy file
- proactiveSecurity : security policy for application and runtime. This element has two children :
  - applicationPolicy (relative path) : path to Java security policy file that will be applied on the application's objects deployed at runtime, like nodes and active objects
  - runtimePolicy (relative path) : path to Java security policy file that will be applied on the ProActive Runtime
- log4jProperties (relative path) : path to the Java log4j configuration file
- userProperties (relative path) : path to the Java properties file
- virtualNode (virtual node) : description of a virtual node. There can be any number of such element

The <proactive> element can have the following attributes :

- relpath (path string) : the location of the ProActive installation
- base (one of ‘HOME’, ‘ROOT’) : base location of the ProActive installation : HOME is the user’s home directory, ROOT is the root directory of the system.

A <virtualNode> element can have the following children :

- nodeProvider (reference to a node provider) : the node provider which will provide the ProActive nodes for this virtual node – see below for description.
- technicalServices (technical service) : a technical service specific to this virtual node. There can be any number of such children.

A virtualNode element can also have the following attributes :

- id (string) : a string identifying this virtual node
- capacity (positive integer) : the capacity requested by this virtual node (that is, the total number of nodes it will request from the node providers which are affected to it). If no capacity is specified, then the Virtual Node will try to get as many nodes as possible. A such Virtual Node is called greedy.

A <nodeProvider> within a <virtualNode> can only have <technicalServices> child elements. These describe technical services specific to this node provider. A <nodeProvider> can also have the following attributes :

- refid (string) : the id of the node provider (as defined in the resources element)
- capacity (positive integer or “max”) : the capacity of this ProActive node provider (that is, the number of ProActive nodes which will be requested from it)

## 2.5. FAQ

## 2.6. Tutorial

This tutorial shows how to deploy a grid-enabled application through the GCM standard. It will present the points of view of both the grid administrator and the application developer.

### 2.6.1. For the Grid Administrator : creating a deployment descriptor

The task of a grid administrator is to make a model of his grid resources through a GCM Deployment Descriptor. Several examples are available in the ProActive distribution. The deployment descriptor should represent the resources of the grid. A deployment descriptor has the following XML structure:

```
<environment>
   <descriptorVariable …/>
…
</environment>

<resources>
   <bridge …/>
   <group>
```

```
    <host .../>
    <host .../>
    ...
  </group>
 ...
</resources>

<acquisition>
  <lookup .../>
  <p2p .../>
 ...
</acquisition>

<infrastructure>
  <hosts>
    <host.../>
  </hosts>

  <groups>
    <groupType .../>
  </group>

  <bridges>
    <bridgeType .../>
  </bridges>
 ...
</infrastructure>
```

The elements must be specified in this order. The <environment> and <acquisition> elements can be omitted, while the <resources> and <infrastructure> ones are mandatory. They are the ones which define the model :

1. Infrastructure: this is a flat list of each individual element of the grid: hosts, groups and bridges listed in no particular order.
2. Resources: this is a tree describing the hierarchical relationships between these infrastructure elements. These relationships are defined by:
   - a host being within which group;
   - a group being behind a bridge;
   - a host being directly available.

Let's examine a couple of basic examples. Considering a very simple grid, that is two desktop PCs networked together. Such a setup would be represented as follows (configuration parameters are omitted for the sake of clarity):

```
<resources>
 <hosts>
  <host refid="host1"  -/>
  <host refid="host2"  -/>
 </hosts>
</resources>

<infrastructure>
  <hosts>
  <host id="host1"  -/>
  <host id="host2"  -/>
  </hosts>
</infrastructure>
```

There is no hierarchical relation between the two hosts, so both resources and infrastructure parts are identical (aside of the extra configuration parameters which are omitted here). A slightly more complex example would be a cluster of 12 mono-processor machines running LSF. The representation in GCM Deployment would be as follows:

```
<resources>
    <group refid="LSF_GROUP">
      <host refid="LSF_GROUP_MEMBER" -/>
    </group>
</resources>

<infrastructure>
  <hosts>
    <host id="LSF_GROUP_MEMBER" -/>
  </hosts>

  <groups>
    <lsfGroup id="LSF_GROUP" >
      <resources processorNumber="12" -/>
    </lsfGroup>
  </groups>
</infrastructure>
```

Within the <infrastructure>, the <hosts> part describes the configuration common to the machines in the group. The <groups> part describes the LSF group itself. Finally, the <resources> part describes how they fit together, in this case the host model being within the LSF group.

The next paragraphs go more in depth on the content and usage of each element.

#### 2.6.1.1. Environment element

To allow for a bit of flexibility, it is possible to define variables in a descriptor. The variables can be used in any XML value element. They cannot be used in an XML element name. The <environment> element is where the variables are defined. It is a simple list of <descriptorVariable> elements. For example:

```
<environment>
  <descriptorVariable name="usertype" value="admin" -/>
  <descriptorVariable name="username" value="jsmith" -/>
</environment>
```

This allows the following usage later on in the descriptor: <sshGroup user="${username}" />

#### 2.6.1.2. Resources element

The <resource> element describes the hierarchical structure of the available grid resources. This can be seen as the topology of the grid: which hosts are part of a group, which group is behind a bridge, etc… All the grid resources which are listed in it must be fully defined in the <infrastructure> element. However it doesn't have to hold every element listed in <infrastructure>, it is meant to contain only the subset of resources which are actually used by the deployment.

You can use the following elements to build your grid topology :

1. <host> : this represents a single machine, or more precisely a single configuration. When used within a group, it represents the common configuration of all machines within this group.
2. <group> : this represents a set of machines all sharing a common configuration. Typically a cluster. The configuration is represented through a Host element.
3. <bridge> : this represents a machine which acts as a gateway to one or several other machines. Typically, a front-end for a cluster.

These elements all take a single argument named 'refid'. The value of the argument is the id of the corresponding host/bridge/group element defined in the <infrastructure> element. The topology must be described according to the following rules:

- A host can be at the top level, or in a group element
- A group can be at the top level, or in a bridge element
- A bridge can only be at the top level

For example, the following constructions are correct: Single host:

```
<host refid="A_HOST" -/>
```

Group:

```
<group refid="CLUSTER">
  <host refid="CLUSTER_NODE" -/>
</group>
```

Group behind a bridge:

```
<bridge refid="CLUSTER_FRONT_END" -/>
  <group refid="CLUSTER">
    <host refid="CLUSTER_NODE" -/>
  </group>
</bridge>
```

### 2.6.1.3. Acquisition element

An alternative to the <infrastructure> element, the <acquisition> element describes how resources which are already running can be acquired. It contains two types of elements: <lookup> and <p2p>, in this order. Each element can either have a single occurrence or be omitted. The <lookup> element has the following three attributes:

1. type: one of "RMI", "HTTP", "IBIS";
2. hostlist: a HostList as defined in 5.1.1;
3. port: a positive integer.

The <p2p> element has a single attribute named "nodesAsked", indicating the number of requested nodes. You can set the value to 'MAX' so that the maximum number of available nodes will be allocated to the task.

### 2.6.1.4. Infrastructure element

The <infrastructure> is where you will list the grid resources on which the deployment can take place, in no particular order. Its purpose is to describe how these resources are deployed (i.e. through which protocols). It can have a single child element of each of the following types: <hosts>, <bridges>, <groups>. <bridges> and <groups> may be empty or omitted, but there should be at least one child element in <hosts>.

## 2.6.2. For the Grid Application Developer : creating an application descriptor

While the Deployment Descriptor lists the grid resources, the application descriptor lists the resources the application needs.

The overall structure of an Application Descriptor is as follows :

```
<environment>
   …
</environment>

<application>
…
</application>

<resources>
   <nodeProvider>
     <file …/>
     …
   </nodeProvider>
   …
</resources>
```

The <environment> element is similar to the one in the Deployment Descriptor. The <application> one is where the application itself and the resources it requests are described (see GCM Application descriptor section). Finally, the <resources> element is where you'll make the link between the requested resources and the deployed ones.

The <application> tag can hold either an <executable> or a <proactive> tag. <executable> is for stand-alone applications which you want to run on a grid. <proactive> is for ProActive-based applications. In both cases the requested resources are specified through <nodeProvider> elements. These elements only carry a single 'refid' attribute which points to a corresponding <nodeProvider> element listed in the <resources> element.

### 2.6.2.1. Example of Executable element

A stand-alone executable is very straightforward to describe. You only need to specify one or several <nodeProvider>s and the application will be run on all the physical nodes these providers can yield.

```
 <application>
   <executable>
     <command name="ls">
       <arg>-lh</arg>
       <arg>--sort=time</arg>
       <arg>*</arg>
     </command>
     <nodeProvider refid="COMPANY_LAN" -/>
   </executable>
 </application>
```

```
<resources>
  <nodeProvider id="COMPANY_LAN">
    <file path="deployment.xml" -/>
  </nodeProvider>
</resources>
```

### 2.6.2.2. Example of ProActive element

A ProActive application runs on virtual nodes. These virtual nodes aggregate physical nodes that they fetch from the node provider specified in the virtual node definition. In the following example, a ProActive application defines two virtual nodes ("master" and "slaves"). The first will fetch a single physical node from the COMPANY_LAN node provider. The second will get as many nodes as available (its capacity is set to "MAX") from both the COMPANY_LAN and REMOTE_CLUSTER providers.

```
<application>
  <proactive relpath="Scratch/ProActive/">

    <configuration>
    <!-- ommitted for clarity --->
    </configuration>

    <virtualNode id="master" capacity="1">
      <nodeProvider refid="COMPANY_LAN" -/>
    </virtualNode>

    <virtualNode id="slaves" capacity="max">
      <nodeProvider refid="COMPANY_LAN" -/>
      <nodeProvider refid="REMOTE_CLUSTER" -/>
    </virtualNode>
  </proactive>
</application>

<resources>
  <nodeProvider id="COMPANY_LAN">
    <file path="deployment.xml" -/>
  </nodeProvider>
  <nodeProvider id="REMOTE_CLUSTER">
    <file path="deployment_cluster.xml" -/>
  </nodeProvider>
</resources>
```

## 2.6.3. For the Grid Application Developer : deploying your application on the grid

To deploy your application on the grid, you need to get your application descriptor as a `java.io.File` . You then pass it to `org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment.loadApplicationDescriptor()` which will return a `GCMApplication` object. To actually start the deployment, simply call the `startDeployment()` method.

```
GCMApplication app;

File desc = new File(this.getClass().getResource("MyApplicationDescriptor.xml").getPath());

app = PAGCMDeployment.loadApplicationDescriptor(desc);
app.startDeployment();
```

If needed you may want to also create a VariableContract and set some of its variables, then pass it as 2nd argument to loadApplicationDescriptor() :

```
VariableContractImpl vContract = new VariableContractImpl();
vContract.setVariableFromProgram("HOST_CAPACITY", -"4",
        VariableContractType.DescriptorDefaultVariable);
vContract.setVariableFromProgram("VM_CAPACITY", -"1",
                VariableContractType.DescriptorDefaultVariable);

GCMApplication app;

File desc = new File(this.getClass().getResource("MyApplicationDescriptor.xml").getPath());

app = PAGCMDeployment.loadApplicationDescriptor(desc, vContract);
app.startDeployment();
```

In the case of a stand-alone application, it will simply be deployed on all the available nodes without any special intervention on your side. In the case of a ProActive-based application, there are two ways for an application to handle the deployment process. The simplest one is to call the

```
  public void waitReady();
```

method on your GCMApplication object. As the name of the method indicates, it amounts to "wait until everything is ready". The call will block until all virtual nodes are ready, that is that they have acquired the minimum number of nodes they need. This method should not be used if one of the virtual nodes is "greedy", in which case it will never be in a "ready" state. There's another version of the method with a timeout parameter :

```
  public void waitReady(int timeout) throws ProActiveTimeoutException;
```

However, a more flexible way is for your application to listen to the availability of new nodes on each virtual node, and act accordingly. The method for this is

```
public void subscribeNodeAttachment(Object client, String methodName, boolean withHistory)
```

in GCMVirtualNode . methodName must be the name of a method of client , which prototype is method(GCMVirtualNode node, String virtualNodeName) . This method will be called by the virtual node for each new available node. To get the list of virtual nodes for your GCMApplication, use

```
  public Map<String, GCMVirtualNode> getVirtualNodes();
```

You can also get a specific virtual node if you know its name :

```
  public GCMVirtualNode getVirtualNode(String vnName);
```

# Chapter 3. User guide

This chapter explains the specific features and functionalities of the GCM Implementation.

## 3.1. Architecture Description Language

The Architecture Description Languages (ADL) are a way to describe software and/or system architectures. ADLs facilitate application description without concern for the underlying implementation code and foster code reuse as an effect of decoupling the implementation from the architecture. Architectures created by using ADLs are composed of predefined entities with various connectors that communicate through defined connections. To define an architecture through an ADL, we can use a textual syntax and/or a graphical syntax, possibly associated with a design tool.

This GCM implementation reuses and extends the Fractal ADL Project. For detailed information on Fractal ADL read the Fractal ADL tutorial [http://fractal.objectweb.org/tutorials/adl/index.html] . This mechanism is used to configure and deploy component systems through normalized XML files. Thanks to a specific XML DTD, it specifies a definition for each component of the application. For instance, it usually describes component interfaces, component bindings, component attributes, the subcomponents in the case of a composite component, the virtual node where the component will be deployed, and so on. As it is an extension of the standard Fractal ADL, GCM allows reusing and integrating ProActive-specific features such as distributed deployment using deployment descriptors, active objects, virtual nodes, etc. For example, in the case of virtual nodes the components ADL has to be associated with a deployment descriptor (this is done at parsing time: both files are given to the parser).

Note that because GCM is based on the Fractal ADL, it requires the following libraries which are included in the /lib directory of the ProActive distribution : fractal-adl.jar , dtdparser.jar , ow_deployment_scheduling.jar .

### 3.1.1. Overview

Components are defined in **definition** files with the .fractal extension. Here is a simple example of an ADL file extract from the example Helloworld retrievable at Examples/org.objectweb.proactive.examples.components.helloworld.

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
-"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.helloworld.HelloWorld">
  <interface name="m" role="server" signature=
"org.objectweb.proactive.examples.components.helloworld.Main"/>

  <component name="client" definition=
"org.objectweb.proactive.examples.components.helloworld.ClientImpl"/>
   <component name="server">
     <interface name="s" role="server" signature=
"org.objectweb.proactive.examples.components.helloworld.Service"/>
    <content class="ServerImpl"/>
    <attributes signature=
"org.objectweb.proactive.examples.components.helloworld.ServiceAttributes">
      <attribute name="header" value="-> -"/>
      <attribute name="count" value="1"/>
    </attributes>
    <controller desc="primitive"/>
  </component>

  <binding client="this.m" server="client.m"/>
  <binding client="client.s" server="server.s"/>

  <controller desc="composite"/>

```

```
28:   <virtual-node name="helooworld-node" cardinality="single"/>
29: </definition>
30:
```

Now, here is a detailed description of each lines:

- 1: Classical prologue of XML files.
- 2-3: The syntax of the document is validated against a DTD retrieved from the classpath.
- 5: The **definition** element has a name (which must be the same name that the file's) and inheritance is supported through the attribute 'extends'.
- 6: The **interface** element allows to specify interfaces of the current enclosing component.
- 9-21: Nesting is allowed for composite components and is done by adding other **component** elements. Components can be specified and created in this definition, and these components can themselves be defined here or in other definition files.
- 14: Primitive components specify the **content** element, which indicates the implementation class containing the business logic for this component.
- 15-19: Components can specify a **attributes** element, which allows to initialize attributes of a component.
-

  23-24: The **binding** element specifies bindings between interfaces of components and specifying 'this' as the name of the component refers to the current enclosing component.
- 26: The **controller** elements can have the following 'desc' values: 'composite' or 'primitive'.
- 28: The **virtual-node** element offers distributed deployment information. It can be exported and composed in the exportedVirtualNodes element.

  The component will be instantiated on the virtual node it specified (or the one that it exported). For a composite component, it means it will be instantiated on the (first if there are several nodes mapped) node of the virtual node. For a primitive component, if the virtual node defines several nodes (cardinality='multiple'), there will be as many instances of the primitive component as there are underlying nodes. Each of these instances will have a suffixed name looking like:

  ```
  primiveComponentName-cyclicInstanceNumber-n
  ```

  where primitiveComponentName is the name defined in the ADL.

The syntax is similar to the standard Fractal ADL, and the parsing engine has been extended. Features specific to ProActive are:

- Virtual nodes have a cardinality property: either 'single' or 'multiple'. When 'single', it means the virtual node in the deployment descriptor should contain 1 node ; when 'multiple', it means the virtual node in the deployment descriptor should contain more than 1 node.
- Virtual nodes can be **exported** and **composed** .
- Template components are not handled.
- The validating DTD has to be specified as: classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd

### 3.1.2. Exportation and composition of virtual nodes

Components are deployed on the virtual node that is specified in their definition ; it has to appear in the deployment descriptor unless this virtual node is exported. In this case, the name of the exported virtual node should appear in the deployment descriptor, unless this exported virtual node is itself exported.

When exported, a virtual node can take part in the composition of other exported virtual nodes. The idea is to further extend reusability of existing (and packaged, packaging being a forthcoming feature of Fractal) components.

In the example, the component defined in helloworld-distributed-wrappers.fractal exports the virtual nodes VN1 and VN2:

```
exportedVirtualNodes exportedVirtualNode name='VN1'
composedFrom composingVirtualNode component='client'
name='client-node' -/composedFrom -/exportedVirtualNode
```

```
exportedVirtualNode name='VN2' composedFrom
composingVirtualNode component='server'
name='server-node'/ -/composedFrom -/exportedVirtualNode
/exportedVirtualNodes
```

VN1 is composed of the exported virtual node 'client-node' from the component named client

In the definition of the client component (ClientImpl.fractal), we can see that client-node is an exportation of a virtual node which is also name 'client-node':

```
exportedVirtualNodes exportedVirtualNode
name='client-node' composedFrom composingVirtualNode
component='this' name='client-node'/ -/composedFrom
/exportedVirtualNode -/exportedVirtualNodes -...
virtual-node name='client-node' cardinality='single'/
```

Although this is a simplistic example, one should foresee a situation where ClientImpl would be a prepackaged component, where its ADL could not be modified ; the exportation and composition of virtual nodes allow to adapt the deployment of the system depending on the existing infrastructure. Colocation can be specified in the enclosing component definition (helloworld-distributed-wrappers.fractal):

```
exportedVirtualNodes exportedVirtualNode name='VN1'
composedFrom composingVirtualNode component='client'
name='client-node' composingVirtualNode
component='server' name='server-node'/ -/composedFrom
/exportedVirtualNode -/exportedVirtualNodes
```

As a result, the client and server component will be colocated / deployed on the same virtual node. This can be profitable if there is a lot of communications between these two components.

When specifying 'null' as the name of an exported virtual node, the components will be deployed on the current virtual machine. This can be useful for debugging purposes.

### 3.1.3. Usage

ADL definitions correspond to component factories. ADL definition can be used directly:

```
Factory factory = org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory();
Map context = new HashMap();
Component c = (Component) factory.newComponent("myADLDefinition",context);
```

It is also possible to use the launcher tool, which parses the ADL, creates a corresponding component factory, and instantiates and assembles the components as defined in the ADL, is started from the `org.objectweb.proactive.core.component.adl.Launcher` class:

```
Launcher [-java|-fractal] <definition>
[<itf>] [deployment-descriptor])
```

where [-java|-fractal] comes from the Fractal ADL Launcher (put -fractal for ProActive components, this will be made optional for ProActive components in the next release), <definition> is the name of the component to be instantiated and started, <itf> is the name of its Runnable interface, if it has one, and <deployment-descriptor> the location of the ProActive deployment descriptor to use. It is also possible to use this class directly from its static main method.

## 3.2. Implementation specific API

### 3.2.1. fractal.provider

The API is the same for any Fractal implementation, though some classes are implementation-specific:

The fractal provider class, that corresponds to the fractal.provider parameters of the JVM, is org.objectweb.proactive.core.component.Fractive . The Fractive class acts as:

- a bootstrap component
- a GenericFactory for instantiating new components
- a utility class providing static methods to create collective interfaces

### 3.2.2. Requirements

As this implementation is based on ProActive, several conditions are required (more informations in the ProActive manual):

- the base class for the implementation of a primitive component has to provide a no-argument and preferably an empty constructor.
- asynchronous method calls with transparent futures is a core feature of ProActive (more informations in the ProActive manual), and it allows concurrent processing. Indeed, suppose a caller invokes a method on a callee. This method returns a result on a component. With synchronous method calls, the flow of execution of the caller is blocked until the result of the method called is received. In the case of intensive computations, this can be relatively long. With asynchronous method calls, the caller gets a future object and will continue its tasks until it really uses the result of the method call. The process is then blocked (it is called wait-by-necessity) until the result has effectively been calculated.

  Thus, for asynchronous invocations, return types of the methods provided by the interfaces of the components have to be reifiable (Non-final and serializable class) and methods must not throw exceptions.

### 3.2.3. Content and controller descriptions

When a component is instantiated with the public Component newFcInstance(Type type, Object controllerDesc, Object contentDesc) throws InstantiationException method of the org.objectweb.fractal.api.factory.Factory class, in addition to the type of the component have to be specified the controller description and the content description of the component.

The controller description ( org.objectweb.proactive.core.component.ControllerDescription ) is useful to describe the controllers of components. It allows to define:

- the name of a component.
- the hierarchical type of a component.
- the custom controllers for a component. The configuration of the controllers is described in a properties file whose location can be given as a parameter. The controllers configuration file is simple: it associates the signature of a controller interface with the implementation that has to be used. During the construction of the component, the membrane is automatically constructed with these controllers. The controllers are linked together, and requests targetting a control interface visit the different controllers until they find the suitable controller, and then the request is executed on this controller.

The role of the content description ( org.objectweb.proactive.core.component.ContentDescription ) is to define some informations about a component:

- the classname of the component (the only one information mandatory).
- the constructor parameters of the component (optional).
- the activity as defined in the ProActive model (optional). See the ProActive manual for more informations about activity in ProActive.
- the meta-object factory for the component (optional).

It is also possible to force that there is only one instance of this component when instantiating the component on a given multiple virtual node by using the forceSingleInstance method.

### 3.2.4. Collective interfaces

Collective interactions are an extension to the Fractal model, described in section Section 3.3, “Collective interfaces” , that relies on collective interfaces.

This feature provides collective interactions (1-to-n and n-to-1 interactions between components), namely gathercast and multicast interfaces

### 3.2.5. Priority controller

In order to define Non Functional prioritized requests (useful for instance for life cycle management, reconfiguration, ...), a partial order between each kind of request is available to specify when an incoming request can pass requests already in the queue.

Here are the different priorities availables for the requests:

- F: Functional request. Always goes at the end of the requests queue.
- NF1: Standard Non Functional request. Also always goes at the end of the requests queue.
- NF2: Non Functional prioritized request. Pass the Functional requests into the requests queue but respect the order of the other Non Functional requests.
- NF3: Non Functional most prioritized request. Pass all the other requests into the requests queue.

Thus, for prioritize non functional requests, a new controller, org.objectweb.proactive.core.component.controller.PriorityController , has to be used:

```
public interface PriorityController {

  /**
   * All the possible kind of priority for a request on a component.
   *
   */
  public enum RequestPriority {
      /**
       * Functional priority
       */
      F,
      /**
       * Non-Functional priority
       */
      NF1,
      /**
       * Non-Functional priority higher than Functional priority (F)
       */
      NF2,
      /**
        * Non-Functional priority higher than Functional priority (F) and Non-Functional priority
       * (NF1 and NF2)
       */
      NF3;
  -}

  /**
   * Set priority of all methods named -'methodName' in the interface -'interfaceName' to
   * -'priority'.
   *
   * @param interfaceName
   *            Name of the component interface providing the method
   * @param methodName
   *            Name of the method on which set the priority
   * @param priority
   *            The priority
   * @return true if success, else false
   */
  public void setPriority(String interfaceName, String methodName, RequestPriority priority);

  /**
```

```
 * Set priority of the method named -'methodName' with the signature defined by -'parametersTypes'
 * in the interface -'interfaceName' to -'priority'.
 *
 * @param interfaceName
 *          Name of the component interface providing the method
 * @param methodName
 *          Name of the method on which set the priority
 * @param parametersTypes
 *          The type of the method's parameters signature
 * @param priority
 *          The priority
 * @return true if success, else false
 */
public void setPriority(String interfaceName, String methodName, Class<?>[] parametersTypes,
       RequestPriority priority);

/**
 * Get the priority for a given method.
 *
 * @param interfaceName
 *          Name of the component interface
 * @param methodName
 *          Name of the method
 * @param parametersTypes
 *          The type of the method's parameters signature
 * @return
 */
public RequestPriority getPriority(String interfaceName, String methodName, Class<?>[]
parametersTypes);
}
```

### 3.2.6. Monitor controller

By using the monitor controller, org.objectweb.proactive.core.component.controller.MonitorController, users can retrieve various statistics on components as the average length of the queue of a given method or the last execution time of another method. Thus, with these metrics, users can be informed on the QoS and then decide to do some changes in their application to improve the performance.

After having started the monitoring (Method startMonitoring), for each methods exposed by the server interfaces of a component, this controller will be able to provide an instance of org.objectweb.proactive.core.component.controller.MethodStatistics, which itself provides some statistics related to a given method.

The set of statistics that can be retrieved and the corresponding methods to call are described through the MethodStatistics interface:

```
/**
 * Get the current length of the requests incoming queue related to the monitored method.
 *
 * @return The current number of pending request in the queue.
 */
public int getCurrentLengthQueue();

/**
 * Get the average number of requests incoming queue per second related to the monitored
 * method since the monitoring has been started.
 *
 * @return The average number of requests per second.
 */
public double getAverageLengthQueue();
```

```
/**
 * Get the average number of requests incoming queue per second related to the monitored
 * method in the last past X milliseconds.
 *
 * @param pastXMilliseconds The last past X milliseconds.
 * @return The average number of requests per second.
 */
public double getAverageLengthQueue(long pastXMilliseconds);

/**
 * Get the latest service time for the monitored method.
 *
 * @return The latest service time in milliseconds.
 */
public long getLatestServiceTime();

/**
 * Get the average service time for the monitored method since the monitoring has been started.
 *
 * @return The average service time in milliseconds.
 */
public double getAverageServiceTime();

/**
 * Get the average service time for the monitored method during the last N method calls.
 *
 * @param lastNRequest The last N method calls.
 * @return The average service time in milliseconds.
 */
public double getAverageServiceTime(int lastNRequest);

/**
 * Get the average service time for the monitored method in the last past X milliseconds.
 *
 * @param pastXMilliseconds The last past X milliseconds.
 * @return The average service time in milliseconds.
 */
public double getAverageServiceTime(long pastXMilliseconds);

/**
 * Get the latest inter-arrival time for the monitored method.
 *
 * @return The latest inter-arrival time in milliseconds.
 */
public long getLatestInterArrivalTime();

/**
 * Get the average inter-arrival time for the monitored method since the monitoring has
 * been started.
 *
 * @return The average inter-arrival time in milliseconds.
 */
public double getAverageInterArrivalTime();

/**
 * Get the average inter-arrival time for the monitored method during the last
 * N method calls.
 *
```

```
  * @param lastNRequest The last N method calls.
  * @return The average inter-arrival time in milliseconds.
  */
 public double getAverageInterArrivalTime(int lastNRequest);

 /**
  * Get the average inter-arrival time for the monitored method in the last past X
  * milliseconds.
  *
  * @param pastXMilliseconds The last past X milliseconds.
  * @return The average inter-arrival time in milliseconds.
  */
 public double getAverageInterArrivalTime(long pastXMilliseconds);

 /**
  * Get the average permanence time in the incoming queue for a request of the monitored method
  * since the monitoring has been started.
  *
  * @return The average permanence time in the incoming queue in milliseconds.
  */
 public double getAveragePermanenceTimeInQueue();

 /**
  * Get the average permanence time in the incoming queue for a request of the monitored method
  * during the last N method calls.
  *
  * @param lastNRequest The last N method calls.
  * @return The average permanence time in the incoming queue in milliseconds.
  */
 public double getAveragePermanenceTimeInQueue(int lastNRequest);

 /**
  * Get the average permanence time in the incoming queue for a request of the monitored method
  * in the last past X milliseconds.
  *
  * @param pastXMilliseconds The last past X milliseconds.
  * @return The average permanence time in the incoming queue in milliseconds.
  */
 public double getAveragePermanenceTimeInQueue(long pastXMilliseconds);

 /**
   * Get the list of all the method calls (server interfaces) invoked by a given invocation.
   *
   * @return The list of the used interfaces.
   * TODO which kind of information do you need (Interface reference, name, -...?)
   */
 public List<String> getInvokedMethodList();
```

In regard to the MethodStatistics instances, they can be recovered by the following MonitorController's methods:

- `public Map<String, MethodStatistics> getAllStatistics();`

  Which returns every MethodStatistics (i.e. statistics of each methods of each server interfaces of the monitored component) contained into a Map. The key to obtain the MethodStatistics corresponding to a method can be generated thanks to the method org.objectweb.proactive.core.component.controller.MonitorControllerHelper.generateKey which takes as parameters the name of the interface, the name of the method and an array of parameter types of the method.

- `public MethodStatistics getStatistics(String itfName, String methodName)`

  Which returns the MethodStatistics object for the method methodName which belongs to the interface itfName.

- `public MethodStatistics getStatistics(String itfName, String methodName, Class<?>[] parametersTypes)`

  Which returns the MethodStatistics object for the method methodName which takes as parameters the given parameter types and belongs to the interface itfName.

These methods to retrieve MethodStatistics are in "immediate services", i.e when calling one of these methods, the corresponding request will not be enqueued in the component queue as any other request but it will be executed immediately and thus avoiding to spend to much time to obtain the statistics.

### 3.2.7. Stream ports

Stream ports allow to ensure to have components which only have one way communication methods (client side to server side).

Thus, by using the org.objectweb.proactive.core.component.StreamInterface interface as a tag on the java interface definition of a component interface, the GCM implementation will check during the instantiation of a Fractal Interface Type created with createFcItfType(...) method, if all the methods of the given interface, and its parents, return void. If not, an exception is thrown: " org.objectweb.fractal.api.factory.InstantiationException " accordingly to the Fractal specifications.

## 3.3. Collective interfaces

In this chapter, we consider multiway communications - communications to or from several interfaces - and notably parallel communications, which are common in Grid computing.

Our objective is to simplify the design of distributed Grid applications with multiway interactions.

The driving idea is to manage the semantics and behavior of collective communications at the level of the interfaces.

### 3.3.1. Motivations

Grid computing uses the resources of many separate computers connected by a network (usually the Internet) to solve large-scale computation problems. Because of the number of available computers, it is fundamental to provide tools for facilitating communications to and from these computers. Moreover, Grids may contain clusters of computers, where local parallel computations can be very efficiently performed - this is part of the solution for solving large-scale computation problems - , which means that programming models for Grid computing should include parallel programming facilities. We address this issue, in the context of a component model for Grid computing, by introducing **collective interfaces** .

The component model that we use, Fractal, proposes two kinds of cardinalities for interfaces, **singleton** or **collection** , which result in one-to-one bindings between client and server interfaces. It is possible though to introduce binding components, which act as brokers and may handle different communication paradigms. Using these intermediate binding components, it is therefore possible to achieve one-to-n, n-to-one or n-to-n communications between components. It is not possible however for an interface to express a collective behavior: explicit binding components are needed in this case.

We propose the addition of new cardinalities in the specification of Fractal interfaces, namely **multicast** and **gathercast** . Multicast and gathercast interfaces give the possibility to **manage a group of interfaces as a single entity** (which is not the case with a collection interface, where the user can only manipulate individual members of the collection), and they **expose** the collective nature of a given interface. Moreover, specific semantics for multiway invocations can be configured, providing users with flexible communications to or from gathercast and multicast interfaces. Lastly, avoiding the use of explicit intermediate binding components simplifies the programming model and type compatibility is automatically verified.

The role and use of multicast and gathercast interfaces are complementary. Multicast interfaces are used for parallel invocations, whereas gathercast interfaces are used for synchronization and gathering purposes.

Note that in our implementation of collective interfaces, new features of the Java language introduced in Java 5 are extensively used, notably annotations and generics.

### 3.3.2. Multicast interfaces

#### 3.3.2.1. Definition

**A multicast interface transforms a single invocation into a list of invocations**

A multicast interface is an abstraction for 1-to-n communications. When a single invocation is transformed into a set of invocations, these invocations are forwarded to a set of connected server interfaces. A multicast interface is unique and it exists at runtime (it is not lazily created). The semantics of the propagation of the invocation and of the distribution of the invocation parameters are customizable (through annotations), and the result of an invocation on a multicast interface - if there is a result - is always a list of results.

Invocations forwarded to the connected server interfaces occur in parallel, which is one of the main reasons for defining this kind of interface: it enables **parallel invocations, with automatic distribution of invocation parameters** .

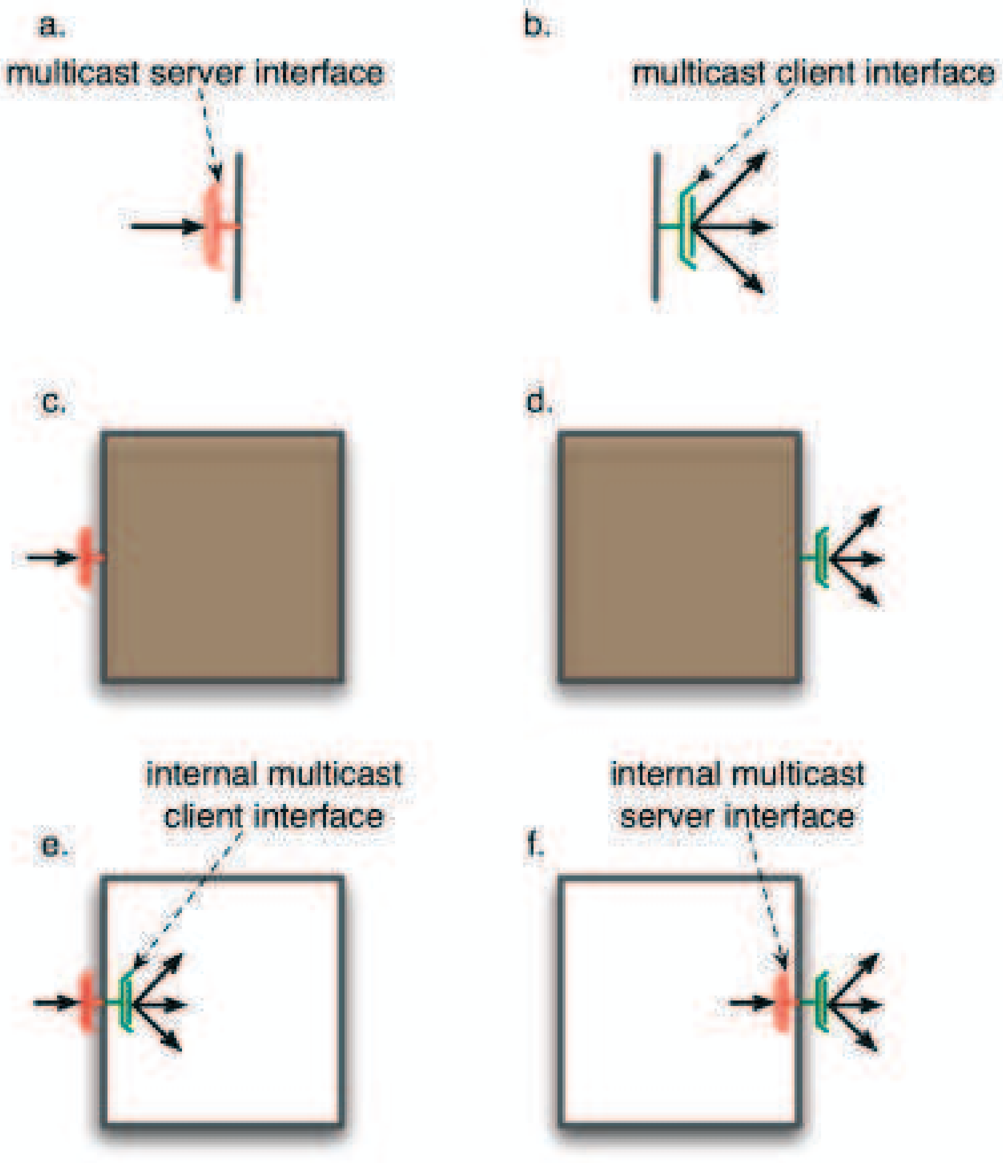

a.
multicast server interface
b.
multicast client interface
c.
d.
internal multicast client interface
e.
internal multicast server interface
f.

**Figure 3.1. Multicast interfaces for primitive and composite component**

### 3.3.2.2. Data distribution

A multicast invocation leads to the invocation services offered by one or several connected server interfaces, with possibly distinct parameters for each server interface.

If some of the parameters of a given method of a multicast interface are lists of values, these values can be distributed in various ways through method invocations to the server interfaces connected to the multicast interface. The default behavior - namely **broadcast** - is to send the same parameters to each of the connected server interfaces. In the case some parameters are lists of values, copies of the lists are sent to each receiver. However, similar to what SPMD programming offers, it may be adequate to strip some of the parameters so that the bound components will work on different data. In MPI for instance, this can be explicitly specified by stripping a data buffer and using the **scatter** primitive.

The following figure illustrates such distribution mechanisms: broadcast (a.) and scatter (b.)

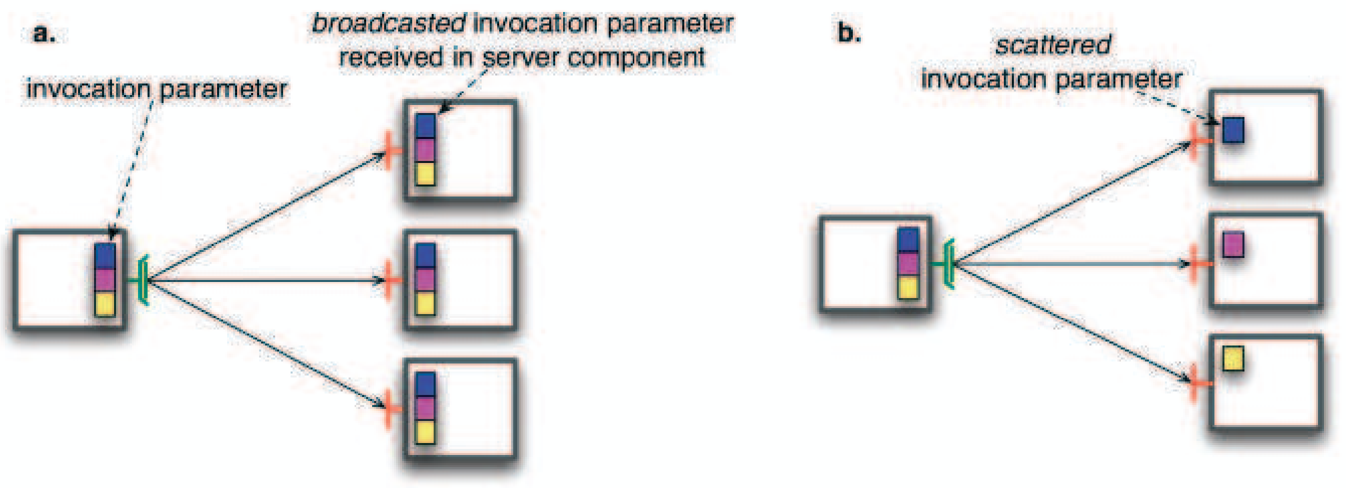

a.
broadcasted invocation parameter received in server component
invocation parameter
b.
scattered invocation parameter

**Figure 3.2. Broadcast and scatter of invocation parameters**

Invocations occur in parallel and the distribution of parameters is automatic.

#### 3.3.2.2.1. Invocation parameters distribution modes

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

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

```
ParamDispatchMode.BROADCAST
```

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

```
ParamDispatchMode.ONE_TO_ONE
```

- ROUND-ROBIN, which distributes each element of the list parameter in a round-robin fashion to the connected server interfaces.

```
ParamDispatchMode.ROUND_ROBIN
```

- RANDOM, which distributes each element of the list parameter in a random manner to the connected server interfaces.

```
ParamDispatchMode.RANDOM
```

- UNICAST, which sends only one parameter of the list parameters to one of the connected server interfaces. .

```
ParamDispatchMode.UNICAST
```

By default, the behavior is not specified: there is no way to predict which parameter will be sent to which server interface. Therefore, it is strongly recommended to combine the use of the UNICAST parameter distribution mode with the dispatch annotation, org.objectweb.proactive.core.group.Dispatch , which allows to specify a custom dispatch mode (This custom dispatch mode has to implement the org.objectweb.proactive.core.group.DispatchBehavior interface):

```
@DispatchMode(mode = DispatchMode.CUSTOM, customMode=CustomUnicastDispatch.class)
```

It is also possible to define a custom partition by specifying the partition algorithm in a class which implements the org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatch interface.

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

### 3.3.2.2.2. Configuration through annotations

Note that our implementation of collective interfaces extensively uses new features of the Java language introduced in Java 5, such as generics and annotations.

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 in the next section. 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);
```

Moreover, an another feature, inherited from the group framework of ProActive, is available: the dynamic dispatch.

The `org.objectweb.proactive.core.group.DispatchMode.DYNAMIC` mode is applicable when there are more parameter partitions, resulting from the use of a parameters distribution mode, than binded server interfaces. Dynamic dispatch uses a knowledge-based policy, i.e. it collects information about request execution by server interfaces, and maintains a ranking among server interfaces so that partitions are dispatched to the "best" server interface. A buffer can be configured, in which case buffered partitions are statically allocated to server interfaces, according to the static dispatch policy.

Dynamic dispatch must be used like this:

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

### 3.3.2.2.3. Results

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, which is just an aggregation of elements of type `T` into a structure of type `list<T>` .

This implies that, for the multicast interface, the signature of the invoked method has to explicitly specify `List<T>` as a return type. This also implies that each method of the interface returns either nothing, or a list. Valid return types for methods of multicast interfaces are illustrated as follows:

```
public List<Something> foo();

public void bar();
```

Otherwise, there is also a possibility to customize the result values by processing a reduction on them. This mechanism allows to gather results and/or perform some operations on them.

There is one reduction mechanism provides by default: SELECT_UNIQUE_VALUE. It allows to extract of the list of results the only one result that the list contains. In order to use it, the multicast interface must use the `org.objectweb.proactive.core.component.type.annotations.multicast.Reduce` annotation at the level of the methods which the results need to be reduced:

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

Or else, a custom reduce mode can also be used. For this case, the first step is to defined the reduction algorithm into a class which implements the `org.objectweb.proactive.core.component.type.annotations.multicast.ReduceBehavior` interface. Then, the multicast interface can use the Reduce annotation, always at the level of the methods, by specifying the mode (CUSTOM) and the implementation class of the reduction to use:

```
@Reduce(reductionMode = ReduceMode.CUSTOM,
customReductionMode = MyReduction.class)
```

### 3.3.2.3. Binding compatibility

Multicast interfaces manipulate lists of parameters (say, List<ParamType> ), and expect lists of results (say, List<ResultType> ). With respect to a multicast interface, connected server interfaces, on the contrary, may work with lists of parameters ( List<ParamType ), but also with individual parameters ( ParamType ) and return individual results ( ResultType ).

Therefore, **the signatures of methods differ from a multicast client interface to its connected server interfaces** . This is illustrated in the following figure: in a. the foo method of the multicast interface returns a list of elements of type T collected from the invocations to the server interfaces, and in b. the bar method distributes elements of type A to the connected server interfaces.

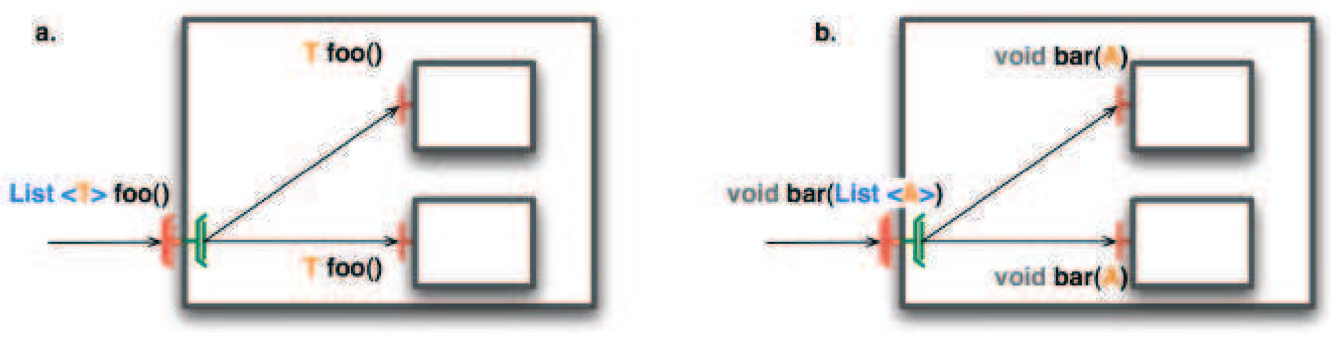


**Figure 3.3. Comparison of signatures of methods between client multicast interfaces and server interfaces.**

For a given multicast interface, the type of server interfaces which may be connected to it can be infered by applying the following rules: for a given multicast interface,

- the server interface must have the same number of methods
- for a given method method foo of the multicast interface, there must be a matching method in the server interface:
  - named foo
  - which returns:
    - void if the method in the multicast method returns void
    - T if the multicast method returns list<T>
  - for a given parameter List<T> in the multicast method, there must be a corresponding parameter, either List<T> or T, in the server interface, which matches the distribution mode for this parameter.

The compatibility of interface signatures is verified automatically at binding time, resulting in a documented IllegalBindingException if signatures are incompatible.

## 3.3.3. Gathercast interfaces

### 3.3.3.1. Definition

**A gathercast interface transforms a list of invocations into a single invocation**

A gathercast interface is an abstraction for n-to-1 communications. It handles data aggregation for invocation parameters, as well as process coordination. It gathers incoming data, and can also coordinate incoming invocations before continuing the invocation flow, by defining synchronization barriers.

Gathering operations require knowledge of the participants of the collective communication (i.e. the clients of the gathercast interface). Therefore, the binding mechanism, when performing a binding to a gathercast interface, provides references on client interfaces bound to the gathercast interface. This is handled transparently by the framework. As a consequence, bindings to gathercast interfaces are bidirectional links.

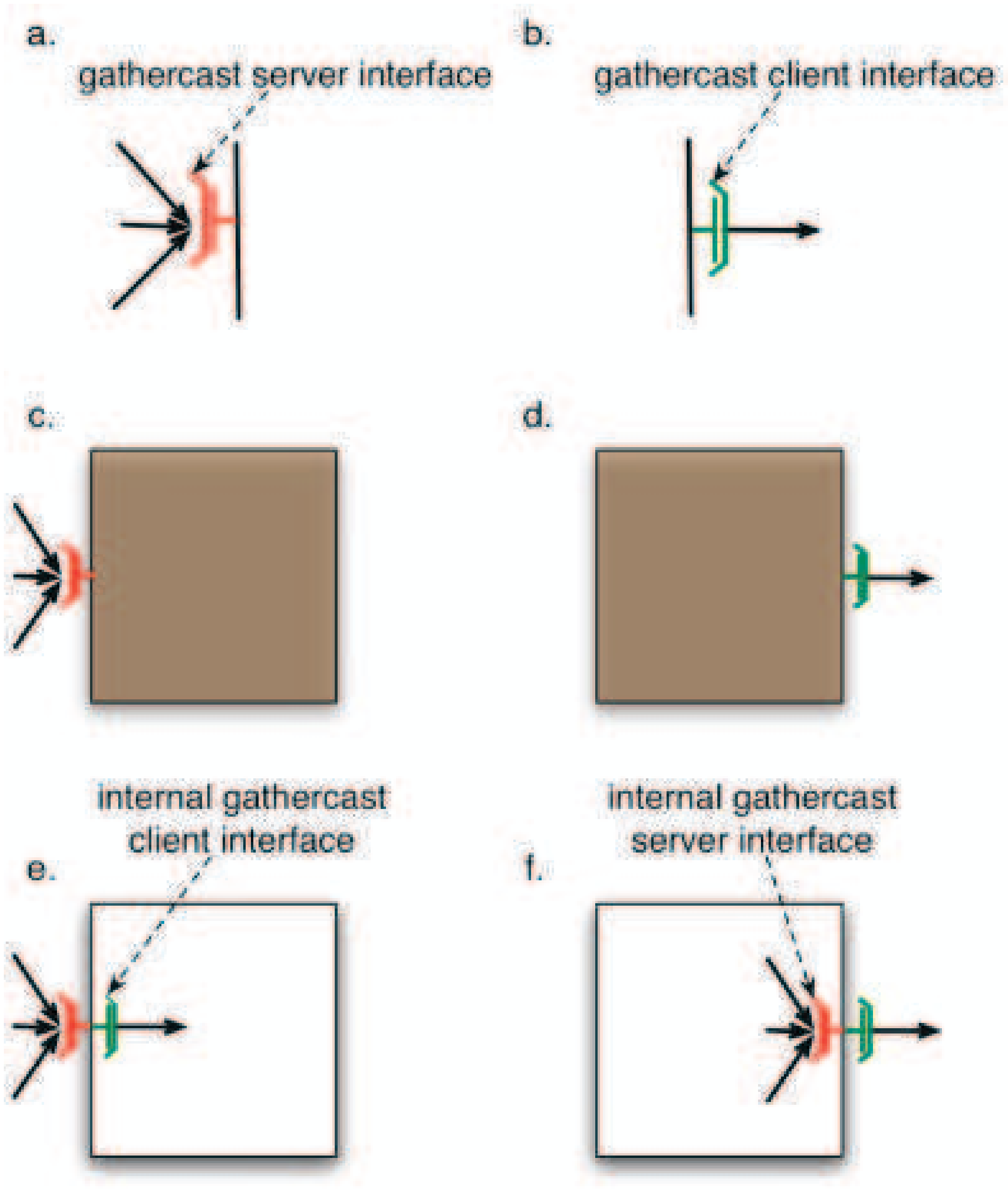

a.
gathercast server interface
b.
gathercast client interface
c.
d.
e.
internal gathercast client interface
f.
internal gathercast server interface

**Figure 3.4. Gathercast interfaces for primitive and composite components**

### 3.3.3.2. Data distribution

Gathercast interfaces aggregate parameters from method invocations from client interfaces into lists of invocations parameters, and they redistribute results to each client interface.

#### 3.3.3.2.1. Gathering of invocation parameters

Invocation parameters are simply gathered into lists of parameters. The indexes of the parameters in the list correspond the index of the parameters in the list of connected client interfaces, managed internally by the gathercast interface.

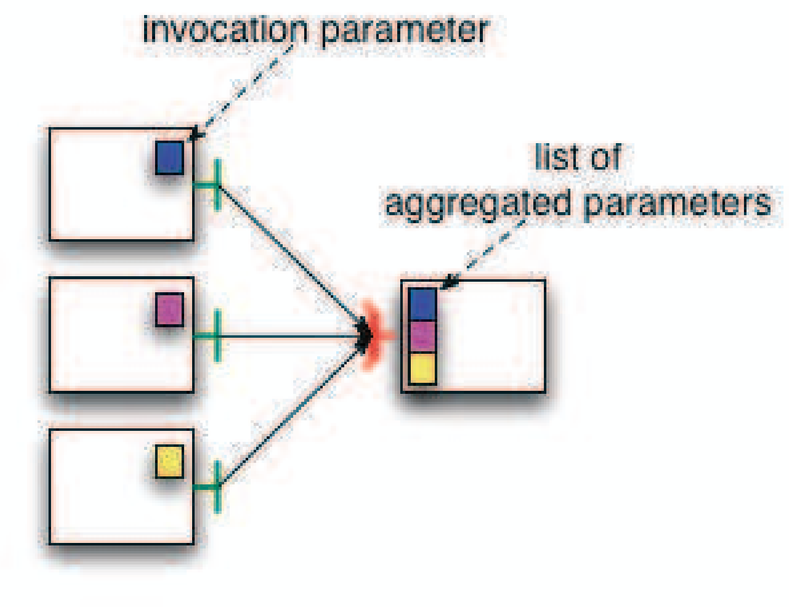


**Figure 3.5. Aggregation of parameters with a gathercast interface**

#### 3.3.3.2.2. Redistribution of results

The result of the invocation transformed by the gathercast interface is a list of values. Each result value is therefore indexed and redistributed to the client interface with the same index in the list of client interfaces managed internally by the gathercast interface.

Similarly to the distribution of invocation parameters in multicast interfaces, a redistribution function could be applied to the results of a gathercast invocation, however this feature is not implemented yet.

### 3.3.3.3. Binding compatibility

Gathercast interfaces manipulate lists of parameters (say, List<ParamType> ), and return lists of results (say, List<ResultType> ). With respect to a gathercast interface, connected client interface work with parameters which can be contained in the lists of parameters of the methods of the bound gathercast interface (ParamType), and they return results which can be contained in the lists of results of the methods of the bound gathercast interface (ResultType).

Therefore, by analogy to the case of multicast interfaces, **the signatures of methods differ from a gathercast server interface to its connected client interfaces** . This is illustrated in the following figure: the foo method of interfaces which are client of the gathercast interface exhibit a parameter of type V , the foo method of the gathercast interface exhibits a parameter of type List<V> . Similarly, the foo method of client interfaces return a parameter of type T , and the foo method of the gathercast interface returns a parameter of type List<T> .

The compatibility of interface signatures is verified automatically at binding time, resulting in a documented IllegalBindingException if signatures are incompatible

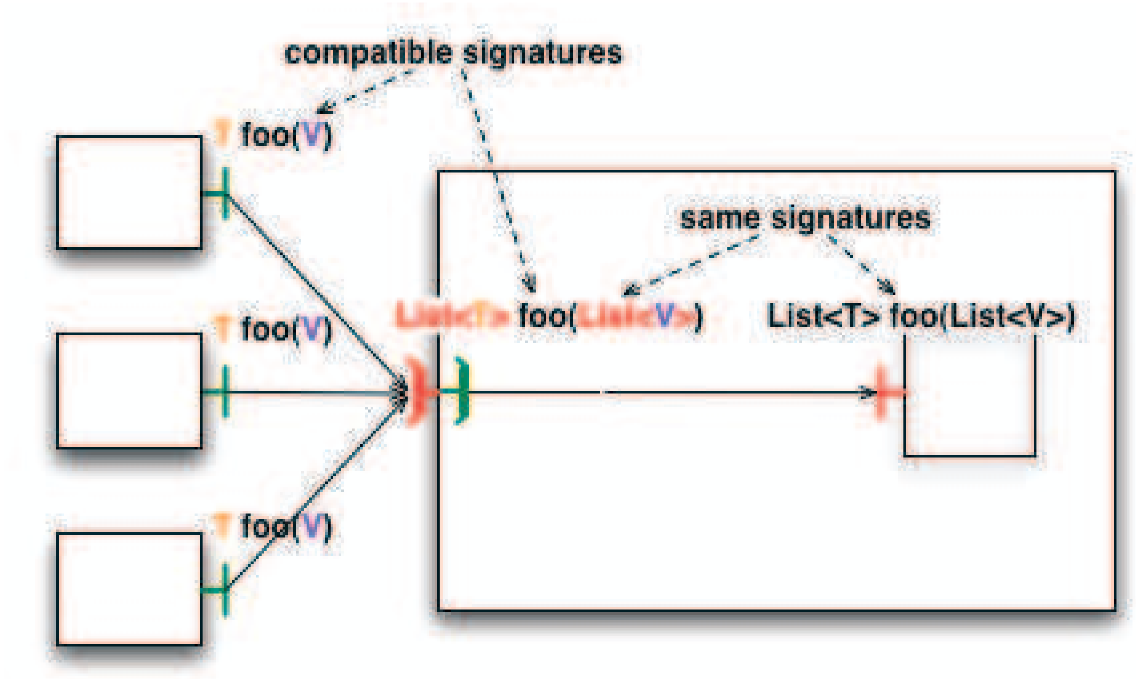


**Figure 3.6. Comparison of signature of methods for bindings to a gathercast interface**

### 3.3.3.4. Process synchronization

An invocation from a client interface to a gathercast interface is asynchronous, provided it matches the usual conditions for asynchronous invocations in ProActive, however the gathercast interface only creates and executes a new invocation with gathered parameters when all connected client interfaces have performed an invocation on it.

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.MethodSynchro(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.

It is also possible for gathercast interface not to wait for all invocations from connected client interfaces to perform an invocation by specifying the waitForAll attribute. Therefore, the gathercast interface will create and execute a new invocation on the first invocation received from any of the connected client interfaces.

Thus, this specific feature can be used by the same annotation as for the timeout but with a different attribute:

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

Therefore, the waitForAll attribute accepts boolean values and has for default value "true" (same behavior as if the annotation is not specified).

Furthermore, it is forbidden to combine timeout and waitForAll set to false (an `org.objectweb.fractal.api.factory.InstantiationException` is raised) because il would be incoherent.

## 3.4. Deployment

### 3.4.1. Overview

To create a distributed component system, a deployment framework is available: the GCM Deployment.

Distribution is achieved in a transparent manner over the Java RMI protocol thanks to the use of a stub/proxy pattern. Components are manipulated indifferently of their location (local or on a remote JVM). A complete description of the GCM Deployment can be found at Chapter 2, *ProActive Grid Component Model Deployment*.

In brief, this framework:

- connects to remote hosts using supported protocols, such as rsh, ssh, lsf, oar, etc...
- creates JVMs on these hosts
- instantiates components on these newly created JVMs

### 3.4.2. Initiate the deployment

The first step to distribute components is to initiate the deployment by loading the Application Descriptor:

```
GCMApplication gcma = PAGCMDeployment.loadApplicationDescriptor(filePath);
```

Then, the deployment must be started (i.e. creation of the remote JVMs):

```
gcma.startDeployment();
```

The next step, distribute components, may be done through the ADL or the API.

### 3.4.3. Distribute components with ADL

Distribute components through the ADL is quite simple:

- Put the GCMApplication into a **java.util.Map** with as key "deployment-descriptor":

```
Map<String, Object> context = new HashMap<String, Object>();
context.put("deployment-descriptor", gcma);
```

- Call the usual method to instantiate component through ADL (method **org.objectweb.fractal.adl.Factory.newComponent**) with the Map containing the Application Descriptor as parameter:

```
Component component = (Component)
    factory.newComponent("my.adl.folder.ComponentDefinition", context);
```

Thus, the component will be instantiated in a node of the virtual node specified in the ADL component definition (if a virtual node of the same name has also been defined in the Application Descriptor).

If no virtual node has been specified in the ADL component definition, the component is created in the local JVM.

### 3.4.4. Distribute components with API

To distribute components through the API, the first thing to do is to get a node from a virtual node defined by the Application Descriptor:

```
Map<String, -? extends GCMVirtualNode> vns = gcma.getVirtualNodes();
Node node = vns.get("VN1").getANode();
```

Then, the component must be instanced thanks to one of the methods provided by **org.objectweb.proactive.core.component.factory.ProActiveGenericFactory**, taking as parameter the node obtained previously:

```
Component component = genericFact.newFcInstance(componentType, controllerDescription,
contentDescription, node);
```

### 3.4.5. ProActive Deployment

The distribution of components may also be made by using the ProActive deployment framework. This deployment framework also uses the concept of virtual nodes but just needs a single configuration file. More informations are available in the ProActive manual.

## 3.5. Advanced

### 3.5.1. Controllers and interceptors

This section explains how to customize the membranes of component through the configuration, composition and creation of controllers and interceptors.

#### 3.5.1.1. Configuration of controllers

It is possible to customize controllers, by specifying a control interface and an implementation.

Controllers are configured in a simple XML configuration file, which has the following structure:

```
<componentConfiguration>
  <controllers>
    <controller>
      <interface>
        ControllerInterface
      </interface>
      <implementation>
        ControllerImplementation
      </implementation>
    </controller>
    -...
```

Unless they some controllers are also interceptors (see later on), the controllers do not have to be ordered.

A default configuration file is provided, it defines the default controllers available for every ProActive component (super, binding, content, naming, lifecycle and component parameters controllers).

A custom configuration file can be specified (in this example with "thePathToMyConfigFile") for any component in the controller description parameter of the newFcInstance method from the Fractal API:

```
componentInstance = componentFactory.newFcInstance(myComponentType,
       new ControllerDescription("name",myHierarchicalType,thePathToMyControllerConfigFile),
       myContentDescription);
```

#### 3.5.1.2. Writing a custom controller

The controller interface is a standard interface which defines which methods are available.

When a new implementation is defined for a given controller interface, it has to conform to the following rules:

1. The controller implementation must extend the AbstractProActiveController class, which is the base class for component controllers in ProActive, and which defines the constructor AbstractProActiveController(Component owner).
2. The controller implementation must override this constructor:

```
public ControllerImplementation(Component owner) {
 super(owner);
}
```

1. The controller implementation must also override the abstract method setControllerItfType(), which sets the type of the controller interface:

```
protected void setControllerItfType() {
 try {
  setItfType(ProActiveTypeFactory.instance().createFcItfType(
   "Name of the controller", TypeFactory.SINGLE));
 }
 catch (InstantiationException e) {
  throw new ProActiveRuntimeException("cannot create controller type: -"
   + this.getClass().getName());
 }
}
```

1. The controller interface and its implementation have to be declared in the component configuration file.

### 3.5.1.3. Configuration of interceptors

Controllers can also act as interceptors: they can intercept incoming invocations and outgoing invocations. For each invocation, pre and post processings are defined in the methods beforeInputMethodInvocation, afterInputMethodInvocation, beforeOutputMethodInvocation, and afterOutputMethodInvocation. These methods are defined in the interfaces InputInterceptor and OutputInterceptor, and take a MethodCall object as an argument. MethodCall objects are reified representations of method invocations, and they contain Method objects, along with the parameters of the invocation.

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:

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

Interceptors can be composed in a basic manner: sequentially.

For input interceptors, the beforeInputMethodInvocation method is called sequentially for each controller in the order 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.

If in the controller configuration file, the list of input interceptors is in this order (the order in the controller configuration file is from top to bottom):

```
InputInterceptor1 InputInterceptor2
```

This means that an invocation on a server interface will follow this path:

```
--> caller ---> InputInterceptor1.beforeInputMethodInvocation --->
 InputInterceptor2.beforeInputMethodInvocation ---> callee.invocation --->
 InputInterceptor2.afterInputMethodInvocation --->
 InputInterceptor1.afterInputMethodInvocation
```

For output interceptors, the beforeOutputMethodInvocation method is called sequentially for each controller in the order they are defined in the controllers configuration file. The afterOutputMethodInvocationmethod is called sequentially for each controller in the reverse order they are defined in the

controllers configuration file.

If in the controller configuration file, the list of input interceptors is in this order (the order in the controller configuration file is from top to bottom):

```
OutputInterceptor1 OutputInterceptor2
```

This means that an invocation on a server interface will follow this path

```
--> currentComponent ---> OutputInterceptor1.beforeOutputMethodInvocation --->
 OutputInterceptor2.beforeOutputMethodInvocation ---> callee.invocation --->
 OutputInterceptor2.afterOutputMethodInvocation --->
 OutputInterceptor1.afterOutputMethodInvocation
```

#### 3.5.1.4. Writing a custom interceptor

An interceptor being a controller, it must follow the rules explained above for the creation of a custom controller.

Input interceptors and output interceptors must implement respectively the interfaces InputInterceptor and OutputInterceptor, 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);
 }

 protected void setControllerItfType() {
  try {
   setItfType(ProActiveTypeFactory.instance().createFcItfType("mycontroller",
             MyController.class.getName(),
             TypeFactory.SERVER,
             TypeFactory.MANDATORY,
             TypeFactory.SINGLE));
  }
  catch(InstantiationException e) {
   throw new ProActiveRuntimeException("cannot create controller" + this
.getClass().getName());
  }
```

```
    }

    // 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 configuration file would state:

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

## 3.5.2. Exporting components as Web Services

As with any active objects, each component can be exposed as a web service.

For a complete description of Web Services in ProActive, please refer to the ProActive manual.

### 3.5.2.1. Using ProActive Web Services API

The main difference with active objects is that, when exposing a component as web service, all the methods of each server interfaces are automatically exposed as a web service (with active objects you can expose just one method at once). Thus, the way to expose a component as web services is quite simple and can be done in a single call method by using **org.objectweb.proactive.extensions.webservices.WebServices** API. After having defined the component and having started it as usually, the following method must be called:

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

where:

- **component** is the instance of the component whose interfaces will be exposed as web services.
- **url** is the url of the web server; typically http://localhost:8080
- **componentName** is the name of the component. Each service available in this way will get a name composed by the component name followed by the interface name: **componentName_interfaceName**.

Likewise, to undeploy a component, just call the method:

```
WebServices.unExposeAsWebService ( String componentName -, String url,  Component component);
```

where:

- **componentName** is the name of the component.

  **url** is the url of the web server; typically http://localhost:8080

  **component** is the instance of the component.

Once the interfaces component are deployed, you can access it via any web service enabled client.

### 3.5.2.2. A simple example: Hello World with component

Here is a very basic example: just a classic HelloWorld programming with component.

First, below is the only one server interface of the component:

```
public interface HelloWorldItf {

  public String helloWorld(String name);
}
```

The implementation class of the component which also contains the main method to create the component, start it and deploy it:

```
public class HelloWorldComponent implements HelloWorldItf {
  public HelloWorldComponent() {
  -}

  public String helloWorld(String name) {
     return "Hello -" + name + " -!";
  -}

  public static void main(String[] args) {
     String url;
     if (args.length == 0) {
        url = "http://localhost:8080";
     -} else {
        url = args[0];
     -}
     if (!url.startsWith("http://")) {
        url = "http://" + url;
     -}
     Component boot = null;
     Component comp = null;
     try {
        boot = org.objectweb.fractal.api.Fractal.getBootstrapComponent();

        TypeFactory tf = Fractal.getTypeFactory(boot);
        GenericFactory cf = Fractal.getGenericFactory(boot);

        // type of server component
        ComponentType sType = tf.createFcType(new InterfaceType[] { tf.createFcItfType(
"hello-world",
              HelloWorldItf.class.getName(), false, false, false) -});
        // create server component
        comp = cf.newFcInstance(sType, new ControllerDescription("server", Constants.PRIMITIVE),
              new ContentDescription(HelloWorldComponent.class.getName()));
```

```
      //start the component
      Fractal.getLifeCycleController(comp).startFc();
    -} catch (InstantiationException e1) {
      e1.printStackTrace();
    -} catch (NoSuchInterfaceException e) {
      e.printStackTrace();
    -} catch (IllegalLifeCycleException e) {
      e.printStackTrace();
    -}

    System.out.println("Deploy an hello world service on -: -" + url);

    WebServices.exposeComponentAsWebService(comp, url, "server");
  -}
}
```

The component comp has been deployed as a web service on the web server located at "http://localhost:8080". The accessible service method is server_hello-world_helloWorld.

At last, here is a java program calling the deployed web service:

```
public class WSClientComponent {
  public static void main(String[] args) {
    String address;
    if (args.length == 0) {
      address = "http://localhost:8080";
    -} else {
      address = args[0];
    -}
    if (!address.startsWith("http://")) {
      address = "http://" + address;
    -}

    address += WSConstants.SERV_RPC_ROUTER;
    System.err.println("address -" + address);

    String namespaceURI = "server_hello-world";
    String serviceName = "server_hello-world";
    String portName = "helloWorld";

    ServiceFactory factory;
    try {
      factory = ServiceFactory.newInstance();

      Service service = factory.createService(new QName(serviceName));

      Call call = service.createCall(new QName(portName));

      call.setTargetEndpointAddress(address);

      call.setOperationName(new QName(namespaceURI, portName));

      call.addParameter("name", new QName("string"), String.class, ParameterMode.IN);

      call.setReturnType(new QName("string"));

      Object[] inParams = new Object[1];
      inParams[0] = "World";
```

```
            String result = ((String) call.invoke(inParams));
            System.out.println(result);
        -} catch (ServiceException e) {
            e.printStackTrace();
        -} catch (RemoteException e) {
            e.printStackTrace();
        -}
    -}
}
```

### 3.5.3. Lifecycle: encapsulation of functional activity in component lifecycle

In this implementation of the Fractal component model, Fractal components are active objects. Therefore it is possible to redefine their activity. In this context of component based programming, we call an activity redefined by a user a functional activity.

When a component is instantiated, its lifecycle is in the STOPPED state, and the functional activity that a user may have redefined is not started yet. Internally, there is a default activity which handles controller requests in a FIFO order.

When the component is started, its lifecycle goes to the STARTED state, and then the functional activity is started: this activity is initialized (as defined in InitActive), and run (as defined in RunActive).

2 conditions are required for a smooth integration between custom management of functional activities and lifecycle of the component:

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

Control invocations to stop the component will automatically set the isActive() return value to false, which implies that when the functional activity loops on the body.isActive() condition, it will end when the lifecycle of the component is set to STOPPED.

### 3.5.4. Structuring the membrane with non-functional components

Components running in dynamically changing execution environments need to adapt to these environments. In Fractal and GCM (Grid Component Model) component models, adaptation mechanisms are triggered by the non- functional (NF) part of the components. Interactions with execution environments may require complex relationships between controllers. In this section we focus on the adaptability of the membrane.Examples include changing communication protocols, updating security policies, or taking into account new runtime environments in case of mobile components. Adaptability implies that evolutions of the execution environments have to be detected and acted upon, and may also imply interactions with the environment and with other components for realizing the adaptation.

We provide tools for adapting controllers. These tools manage (re)configuration of controllers inside the membrane.For this, we provide a model and an implementation, using a standard component- oriented approach for both the application (functional) level and the control (NF) level. Having a component-oriented approach for the non-functional aspects also allows them to benefit from the structure, hierarchy and encapsulation provided by a component-oriented approach.

In this section, we propose to design NF concerns as compositions of components as suggested in the GCM proposal.Our general objective is to allow controllers implemented as components to be directly plugged in a component membrane. These controllers take advantage of the properties of component systems like **reconfigurability**, i.e. changing of the contained components and their bindings.This allows components to be dynamically adapted in order to suit changing environmental conditions. Indeed, among others, we aim at a component platform appropriate for **autonomic Grid applications**; those appli- cations aim to ensure some quality of services and other NF features without being geared by an external entity.

Components in the membrane introduce two major changes : first, refinements of the Fractal/GCM model concerning the structure of a membrane; second, a definition and an implementation of an API that allows GCM membranes to be themselves composed of components, possibly distributed. Both for efficiency and for flexibility reasons, we provide an implementation where controllers can either be classical objects or full components that could even be distributed. We believe that this high level of flexibility is a great advantage of this approach over the existing ones [8, 7]. Our model refinements also provide a better structure for the membrane and a better decoupling between the membrane and its externals. Finally, our approach gives the necessary tools for membrane reconfiguration, providing flexibility and evolution abilities. The API we present can be split in two parts:

- Methods dedicated to component instantiation: they allow the specification of a NF type of a component, and the instantiation of NF components;
- Methods for the management of the membrane: they consist in managing the content, introspecting , and managing the life-cycle of the membrane. Those methods are proposed as an extension of the Fractal component model, and consequently of the GCM;

### 3.5.4.1. Motivating example

Here we present a simple example that shows the advantages of componen tizing controllers of GCM components. In our example, we are considering a naive solution for securing communications of a composite component. As described in Figure Figure 3.7, “ Example: architecture of a naive solution for secure communications ”, secure communications are implemented by three components inside the membrane: Interceptor, Decrypt, and Alert. The scenario of the example is the following: the composite component receives encrypted messages on its server functional interface. The goal is to decrypt those messages. First, the incoming messages are intercepted by the Interceptor component. It forwards all the intercepted communications to Decrypt, which can be an off-the-shelf component (written by cryptography specialists) implementing a specific decryption algorithm. The Decrypt component receives a key for decryption through the non-functional server interface of the composite (interface number 1 on the figure). If it successfully decrypts the message, the Decrypt component sends it to the internal functional components, using the functional internal client interface (2). If a problem during decryption occurs, the Decrypt component sends a message to the Alert component. The Alert component is charge to decide on how to react when a decryption fails. For example, it can contact the sender (using the non-functional client interface – 3) and ask it to send the message again. Another security policy would be to contact a “trust and reputation” authority to signal a suspicious behaviour of the sender. The Alert component is implemented by a developer who knows the security policy of the system. In this example, we have three well-identified components, with clear functionalities and connected through well-defined interfaces. Thus, we can dynamically replace the Decrypt component by another one, implementing a different decryption algorithm. Also, for changing the security policy of the system, we can dynamically replace the Alert component and change its connexions. Compared to a classical implementation of secure communications (for example with objects), using components brings to the membrane a better structure and reconfiguration possibilities. To summarize, componentizing the membrane in this example provides dynamic adaptability and reconfiguration; but also re-usability and composition from off-the-shelf components.

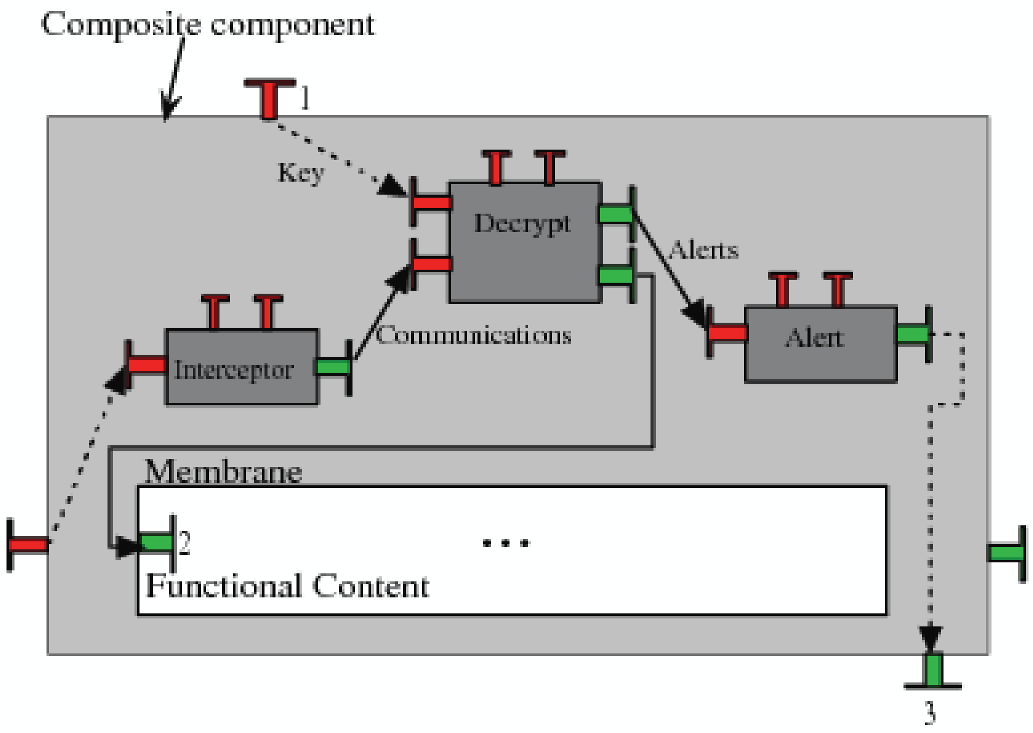
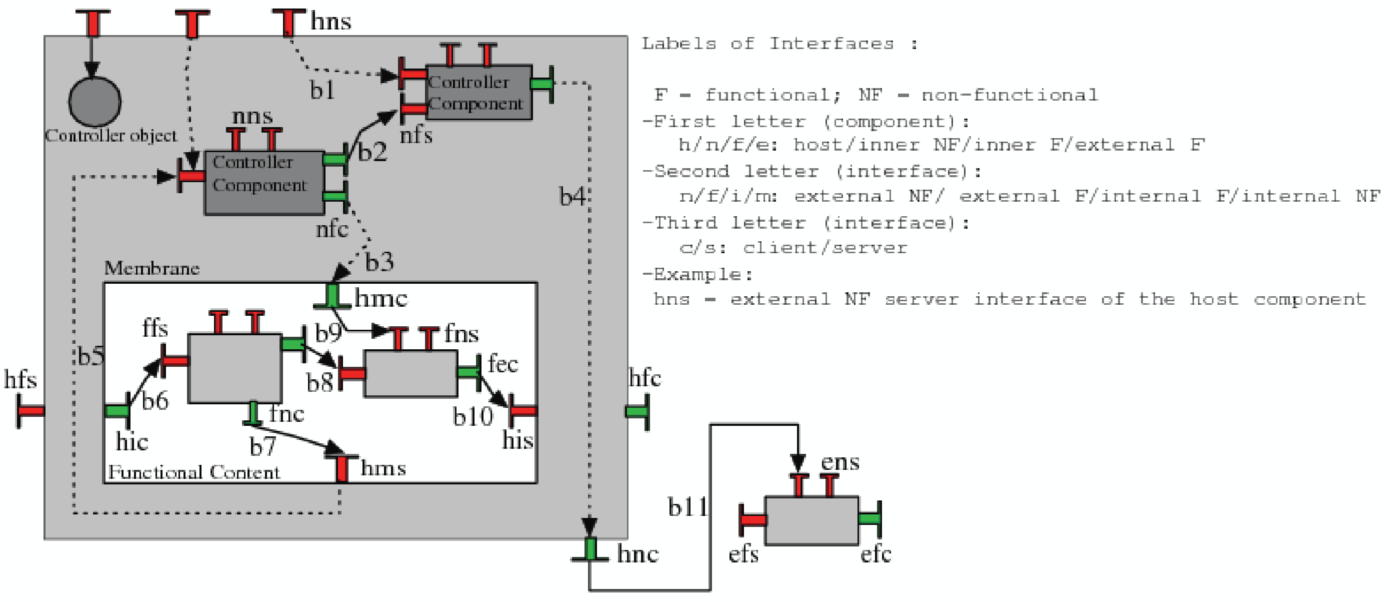


**Figure 3.7. Example: architecture of a naive solution for secure communications**

### 3.5.4.2. A structure for Componentized Membranes



```
Labels of Interfaces :

 F - functional; NF - non-functional
-First letter (component):
   h/n/f/e: host/inner NF/inner F/external F
-Second letter (interface):
   n/f/i/m: external NF/ external F/internal F/internal NF
-Third letter (interface):
   c/s: client/server
-Example:
 hns - external NF server interface of the host component
```

**Figure 3.8. Structure for the membrane of Fractal/GCM components**

Figure Figure 3.8, " Structure for the membrane of Fractal/GCM components " shows the structure we suggest for the component membrane. The membrane (in gray) consists of one object controller and two component controllers, the component controllers are connected together and with the outside of the membrane by different bindings. For the moment, we do not specify whether components are localized with the membrane, or distributed.

Before defining an API for managing components inside the membrane, the definition of the membrane given by the GCM specification needs some refinements. Those refinements, discussed in this section, provide more details about the structure a membrane can adopt. Figure Figure 3.8, " Structure for the membrane of Fractal/GCM components " represents the structure of a membrane and gives a summary of the different kinds of interface roles and bindings a GCM component can provide. As stated in the GCM specification, NF interfaces are not only those specified in the Fractal specification, which are only external server ones. Indeed, in order to be able to compose NF aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality, and contingency. For example, in GCM, client NF interfaces allow for the composition of NF aspects and reconfigurations at the NF level. Our model is also flexible, as all server NF interfaces can be implemented by both objects or components controllers.

All the interfaces showed in Figure Figure 3.8, " Structure for the membrane of Fractal/GCM components " give the membrane a better structure and enforce decoupling between the membrane and its externals. For example, to connect **nfc** with **fns**, our model adds an additional stage: we have first to perform binding **b3**, and then binding **b9**. This avoids **nfc** to be strongly coupled with **fns**: to connect **nfc** to another **fns**, only binding **b9** has to be changed. In Figure Figure 3.8, " Structure for the membrane of Fractal/ GCM components ", some of the links are represented with dashed arrows. Those links are not real bindings but "alias" bindings (e.g. b3); the source interface is the alias and it is "merged" with the destination interface. These bindings are similar to the export/ import bindings existing in Fractal (b6, b10) except that no interception of the communications on these bindings is allowed.

#### 3.5.4.2.1. Performance issues

While componentizing the membrane clearly improves its programmability and its capacity to evolve, one can wonder what happens to performance. First, as our design choice allows object controllers, one can always keep the efficiency of crucial controllers by keeping them as objects. Second, the overhead for using components instead of objects is very low if the controllers components are local, and are negligible compared to the communication time, for example. Finally, if controllers components are distributed, then there can be a significant overhead induced by the remote communications, but if communications are asynchronous, and the component can run in parallel with the membrane, this method can also induce a significant speedup, and a better availability of the membrane. To summarize, controllers invoked frequently and performing very short treatments, would be more efficiently implemented by local objects or local components. For controllers called less frequently or which involve long computations, making them distributed would improve performances and availability of the membrane.

### 3.5.4.3. An API for (Re)configuring Non-functional Aspects

#### 3.5.4.3.1. Non-functional Type and Non-functional Components

To typecheck bindings between membranes, we have to extend the GCM model with a new concept: the non-functional type of a component. This type is defined as the union of the types of NF interfaces the membrane exposes. To specify the NF type of a component, we propose to overload the Fractal newFcInstance method (the one to create functional components) as follows:

```
        public Component newFcInstance(Type fType,Type nfType, any contentDesc, any
controllerDesc);
```

In this method, nfType represents the NF type of the component; it can be specified by hand. Of course the standard Fractal type factory has to be extended in order to support all possible roles of NF interfaces. Soon, it should be possible to specify the NF type within a configuration file: the controller descriptor argument (controllerDesc) can be a file written in Architecture Description Language (ADL) containing the whole description of the NF system.

Components inside the membrane are **non-functional components**. They are similar to functional ones. However, their purpose is different because they deal with NF aspects of the **host component**. Thus, in order to enforce separation of concerns, we restrict the interactions between functional and NF components. For example, a NF component cannot be included inside the functional content of a composite. Inversely, a functional component cannot be added inside a membrane. As a consequence, direct bindings between functional interfaces of NF and functional components are forbidden. To create NF components, we extend the common Fractal factories (generic factory and ADL factory). For generic factory, we add a method named newNFcInstance that creates this new kind of components:

```
        public Component newNFcInstance(Type fType,Type nfType, any contentDesc, any
controllerDesc);
```

Parameters of this method are identical to its functional equivalent and NF components are created the same way as functional ones.

#### 3.5.4.3.2. API for the management of the membrane

```
        public void addNFSubComponent(Component component) throws IllegalContentException;
        public void removeNFSubComponent(Component component) throws IllegalContentException,
IllegalLifeCycleException, NoSuchComponentException;
        public Component[] getNFcSubComponents();
        public Component getNFcSubComponent(string name) throws NoSuchComponentException;
        public void setControllerObject(string itf, any controllerclass) throws
NoSuchInterfaceException;
        public void startMembrane() throws IllegalLifeCycleException;
        public void stopMembrane() throws IllegalLifeCycleException;
```

**Figure 3.9. The primitives for managing the membrane.**

To manipulate components inside membranes, we introduce primitives to perform basic operations like adding, removing or getting a reference on a NF component. We also need to perform calls on well-known Fractal controllers (life-cycle controller, binding controller, . . . ) of these components. So, we extend Fractal/GCM specification by adding a new controller called membrane controller. As we want it to manage all the controllers, it is the only mandatory controller that has to belong to any membrane. It allows the manual composition of membranes by adding the desired controllers. The methods presented in Figure Figure 3.9, “ The primitives for managing the membrane. ” are included in the MembraneController interface; they are the core of the API and are sufficient to perform all the basic manipulations inside the membrane. They add, remove, or get a reference on a NF component. They also allow the management of ob- ject controllers and membrane’s life-cycle. Referring to Fractal, this core API implements a subset of the behavior of the life-cycle and content controllers specific to the membrane. This core API can be included in any Fractal/GCM implementation. Reconfigurations of NF components inside the membrane are performed by calling standard Fractal controllers. The general purpose API defines the following methods:

- addNFSubComponent(Component component): adds the NF component given as argument to the membrane;
- removeNFSubComponent(Component component): removes the specified component from the membrane;
- getNFcSubComponents(): returns an array containing all the NF components;
- getNFcSubComponent(string name): returns the specified NF component, the string argument is the name of the component;
- setControllerObject(string itf, any controllerclass): sets or replaces an existing controller object inside the membrane. Itf specifies the name of the control interface which has to be implemented by the controller class, given as second parameter. Replacing a controller object at runtime provides a very basic adaptivity of the membrane;
- startMembrane(): starts the membrane, i.e. allows NF calls on the host component to be served. This method can adopt a recursive behavior, by starting the life-cycle of each NF component inside the membrane;
- stopMembrane(): Stops the membrane, i.e. prevents NF calls on the host component from being served except the ones on the membrane controller. This method can adopt a recursive behavior, by stopping the life-cycle of each NF component.

#### 3.5.4.3.3. Higher level API

```
        public void bindNFc(String clientItf, String serverItf) throws NoSuchInterfaceException,
IllegalLifeCycleException,IllegalBindingException, NoSuchComponentException;
        public void bindNFc(String clientItf, Object serverItf) throws NoSuchInterfaceException,
IllegalLifeCycleException,IllegalBindingException, NoSuchComponentException;
        public void unbindNFc(String clientItf) throws NoSuchInterfaceException,
IllegalLifeCycleException, llegalBindingException, NoSuchComponentException;
        public String[] listNFc(String component) throws NoSuchComponentException;
        public Object lookupNFc(String itfname) throws
NoSuchInterfaceException,NoSuchComponentException;
        public void startNFc(String component) throws IllegalLifeCycleException,
NoSuchComponentException;
        public void stopNFc(String component) throws IllegalLifeCycleException,
NoSuchComponentException;
        public String getNFcState(String component) throws NoSuchComponentException;
```

**Figure 3.10. Higher level API**

In Figure Figure 3.10, “ Higher level API ”, we present an alternative API, that addresses NF components by their names instead of their references. These methods allow to make calls on the binding controller and on the life-cycle controller of NF components that are hosted by the component membrane. Currently, they don’t take into account the hierarchical aspect of NF components. The method calls address the NF components and call their controllers at once. For example, here is the Java code that binds two components inside the membrane using the general purpose API. It binds the interface “i1” of the component “nfComp1” inside the membrane to the interface “i2” of the component “nfComp2”. Suppose mc is a reference to the MembraneController of the host component.

```
        Component nfComp1=mc.getNFcSubComponent("nfComp1");
        Component nfComp2=mc.getNFcSubComponent("nfComp2");
        Fractal.getBindingController(nfComp1).bindFc("i1",nfComp2.getFcInterface("i2"));
```

Using the API of Figure Figure 3.10, “ Higher level API ”, this binding can be realized by the following code, that binds the component “nfComp1” correctly.

```
        mc.bindNFc("nfComp1.i1","nfComp2.i2");
```

Similarly to the example above, all the methods of Figure Figure 3.10, “ Higher level API ” result in calls on well-known Fractal controllers. Interfaces are represented as strings of the form component.interface, where component is the name of the inner component and interface is the name of its client or server interface. We use the name “mem- brane” to represent the membrane of the host component, e.g. membrane.i1 is the NF interface i1 of the host component; in this case interface is the name of an interface from the NF type. For example, **bindNFc(string, string)** allows to perform the bindings: **b1**, **b2**, **b4**, **b3**, **b9**, **b7** and **b5** of Figure Figure 3.8, “ Structure for the membrane of Fractal/GCM components ”.

The API presented in Figure Figure 3.10, “ Higher level API ” introduced higher level mechanisms for reconfiguring the membrane. It also solves the problem of local components inside the membrane. As usual in distributed programming paradigms, GCM objects/ components can be accessed locally or remotely. Remote references are accessible everywhere, while local refer- ences are accessible only in a restricted address space. When returning a local object/component outside its address space, there are two alternatives: create a remote reference on this entity; or make a copy of it. When considering a copy of a NF local component, the NF calls are not consistent. If an invocation on getNFcSubComponent(string name) returns a copy of the specified NF component, calls performed on this copy will not be performed on the “real” NF component inside the membrane. Methods introduced in Figure Figure 3.10, “ Higher level API ” solve this problem.

## 3.5.5. Short cuts

### 3.5.5.1. Principles

Communications between components in a hierarchical model may involve the crossing of several membranes, 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 shortcutting: communicating directly from a caller component to a callee component by avoiding indirections in the membranes.

In the Julia implementation, a shortcut mechanism is provided for components in the same JVM, and the implementation of this mechanism relies on code generation techniques.

We provide a shortcut mechanism for distributed components, and the implementation of this mechanism relies on a "tensioning" technique: the first invocation determines the shortcut path, then the following invocations will use this shortcut path.

For example, in the following figure, 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 bypassing of the wrapper components, which results in 2 local communications for the sole functional interface.

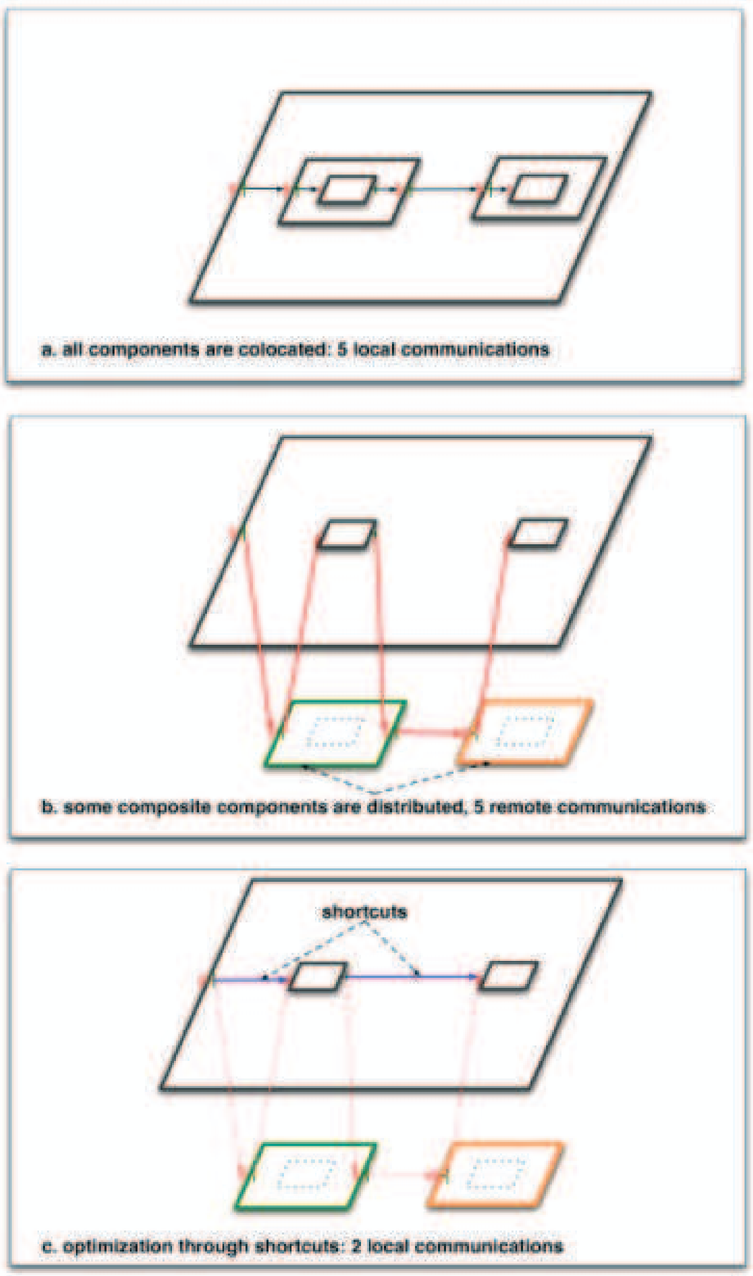

a. all components are colocated: 5 local communications
b. some composite components are distributed, 5 remote communications
shortcuts
c. optimization through shortcuts: 2 local communications

**Figure 3.11. Using shortcuts for minimizing remote communications.**

### 3.5.5.2. Configuration

Shortcuts are available when composite components are synchronous components (this does not break the ProActive model, as composite components are structural components). 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 will automatically establish short cuts between components whenever possible.

# Chapter 4. Architecture and design

The implementation of the Fractal model is achieved by reusing the extensible architecture of ProActive, notably the meta-object protocol and the management of the queue of requests. As a consequence, components are fully compatible with standard active objects and as such, inherit from the features active objects exhibit: mobility, security, deployment etc.

A fundamental idea is to manage the non-functional properties at the meta-level: **each component is actually an active object** with dedicated meta-objects in charge of the component aspects.

## 4.1. Meta-object protocol

ProActive is based on a meta-object protocol (MOP), that allows the addition of many aspects on top of standard Java objects, such as asynchronism and mobility. Active objects are referenced indirectly through stubs: this allows transparent communications, would the active objects be local or remote.

The following diagram explains this mechanism:

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

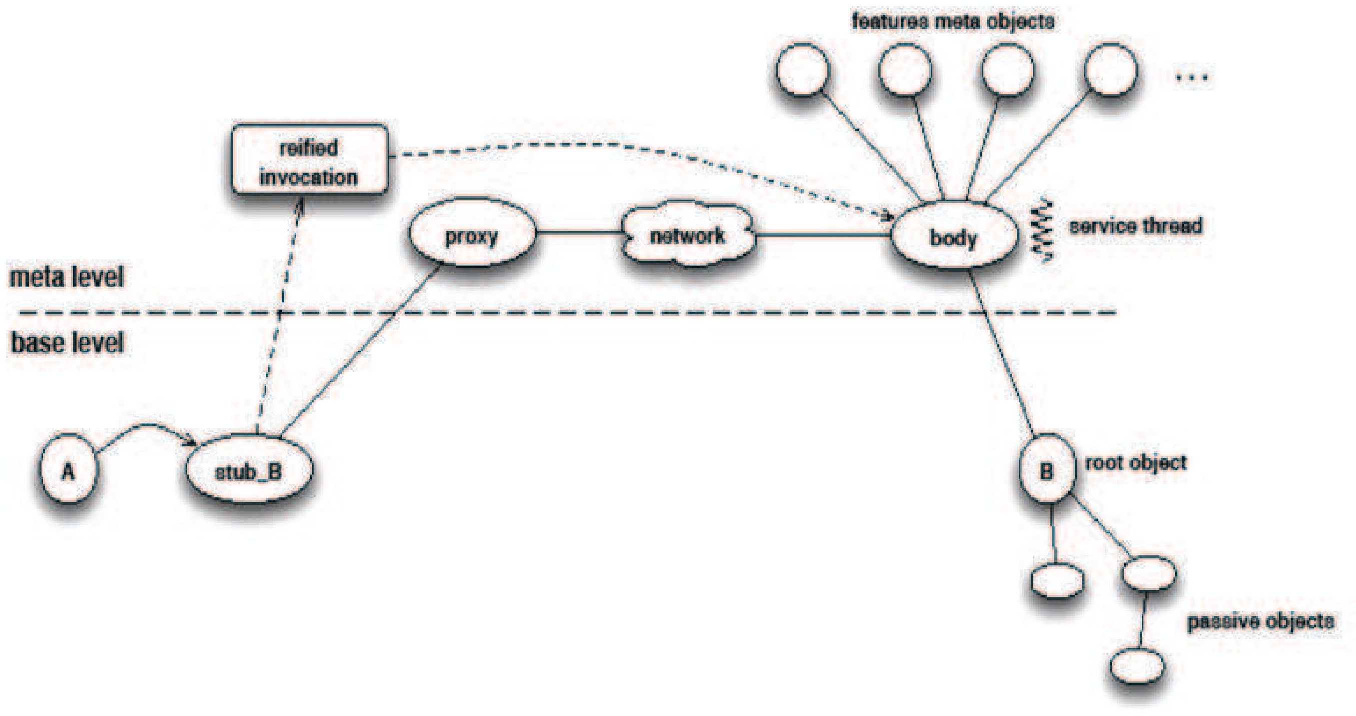


**Figure 4.1. ProActive's Meta-Objects Protocol.**

The same idea is used to manage components: we just add a set of meta-objects in charge of the component aspects.

The following diagram shows what is changed:

A new set of meta-objects, managing the component aspect (constituting the controller of the component, in the Fractal terminology), is added to the active object 'a'. The standard ProActive stub (that gives a representation of type A on the figure) is not used here, as we manipulate components. In Fractal, a reference on a component is of type Component , and references to interfaces are of type Interface . 'b' can now manipulate the component based on 'a' through a specific stub, called a **component representative**

. This **component representative** is of type Component , and also offers references to control and functional interfaces, of type Interface . Note that classes representing functional interfaces of components are generated on the fly: they are specific to each component and can be unknown at compile-time.

Method invocations on Fractal interfaces are reified and transmitted (possibly through a network) to the body of the active object corresponding to the component involved. All standard operations of the Fractal API are now accessible.

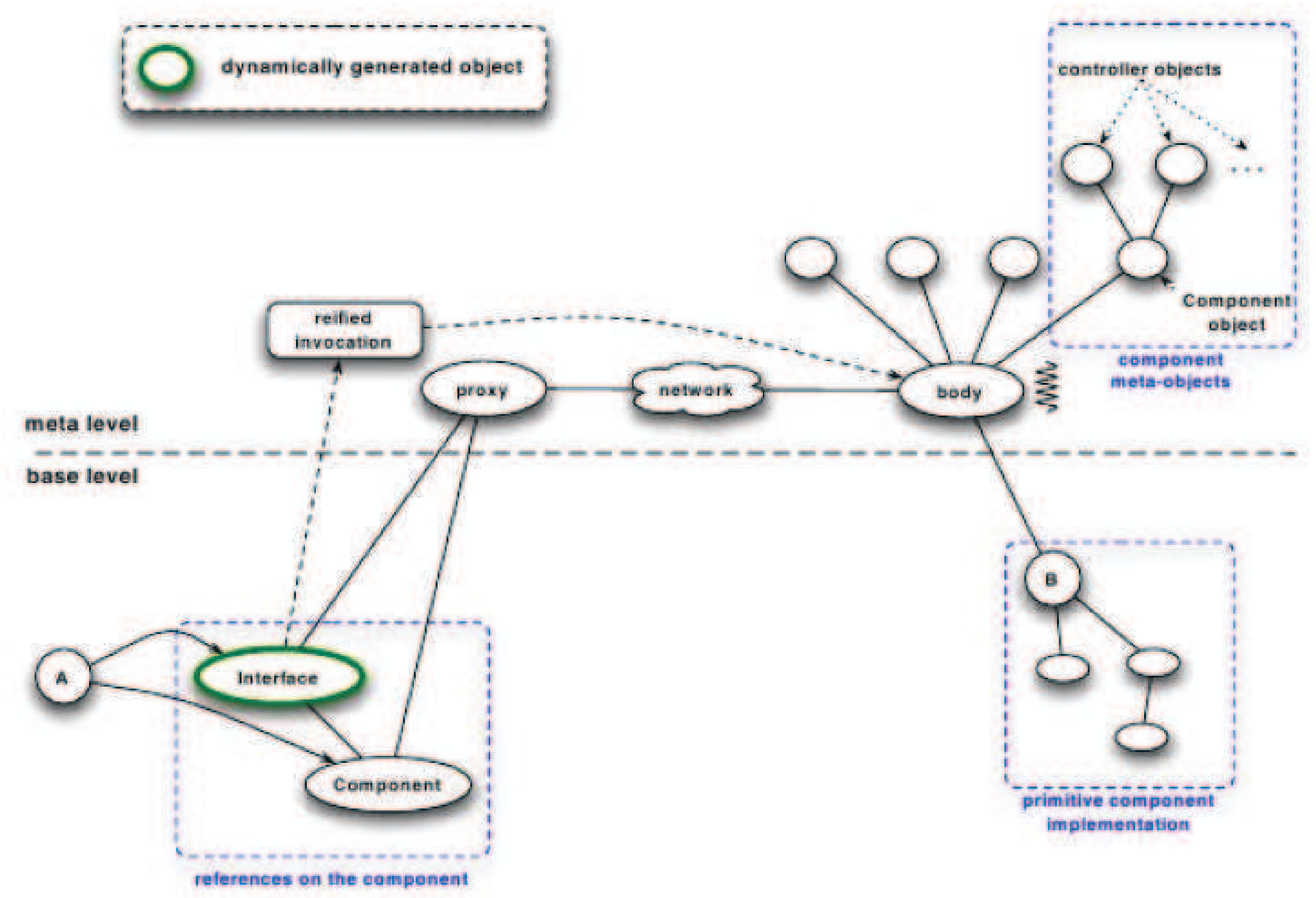


**Figure 4.2. The ProActive MOP with component meta-objects and component representative**

## 4.2. Components vs active objects

In our implementation, because we make use of the MOP's facilities, all components are constituted of one active object (at least), are they composite or primitive components. If the component is a composite, and if it contains other components, then we can say it is constituted of several active objects. Also, if the component is primitive, but the programmer of this component has put some code within it for creating new active objects, the component is again constituted of several active objects.

As a result, a composite component is an active object built on top of the CompositeComponent class, and a parallel component is built on top of the ParallelComponent class. These classes are empty classes, because for composite and parallel components, all the action takes place in the meta-level. But they are used as a base to build active objects, and their names help to identify them with the IC2D visual monitoring tool.

## 4.3. Method invocations on components interfaces

Invoking a method on an active object means invoking a method on the stub of this active object. What usually happens then is that the method call is reified as a Request object and transferred (possibly through a network) to the body of the active object. It is then redirected towards the queue of requests, and delegated to the base object according to a customizable serving policy (standard is FIFO).

Component requests, on the other hand, are tagged so as to distinguish between functional requests and controller requests. A functional request targets a functional interface of the component, while a controller request targets a controller of the component.

Like in the standard case (without components), requests are served from the request queue. The serving policy has to be FIFO to ensure coherency. **This is where the life cycle of the components is controlled** : the dispatching of the request is dependent upon the nature of the request, and corresponds to the following algorithm:

```
loop if componentLifeCycle.isStarted() get next request -//
all requests are served else if
componentLifeCycle.isStopped() get next controller request
// only controller requests are served -; if gotten request
is a component life cycle request if request is start --->
set component state to started -; if request is stop --->
set component state to stopped -; -; -;
```

# Chapter 5. Component examples

Three examples are presented: code snippets for visualizing the transition between active objects and components, the 'hello world', from the Fractal tutorial, and C3D component version. The programming model is Fractal, and one should refer to the Fractal documentation for other detailed examples.

## 5.1. From objects to active objects to distributed components

In Java, objects are created by instantiation of classes. With ProActive, one can create active objects from Java classes, while components are created from component definitions. Let us first consider the 'A' interface:

```
public interface A {
  public String foo(); // dummy method
}
```

'AImpl' is the class implementing this interface:

```
public class AImpl implements A {
 public AImpl() {}
 public String foo() {
 // do something
 -}
}
```

The class is then instantiated in a standard way:

```
 A object = new AImpl();
```

Active objects are instantiated using factory methods from the ProActive class (see the ProActive manual). It is also possible to specify the activity of the active object, the location (node or virtual node), or a factory for meta-objects, using the appropriate factory method.

```
A active_object = (A)PAActiveObject.newActive(
 AImpl, // signature of the base class
 new Object[] {}, // Object[]
 aNode, // location, could also be a virtual node
);
```

As components are also active objects in this implementation, they benefit from the same features, and are configurable in a similar way. Constructor parameters, nodes, activity, or factories, that can be specified for active objects, are also specifiable for components. The definition of a component requires 3 sub-definitions: the type, the description of the content, and the description of the controller.

### 5.1.1. Type

The type of the component (i.e. the functional interfaces provided and required) is specified in a standard way: (as taken from the Fractal tutorial)

We begin by creating objects that represent the types of the components of the application. In order to do this, we must first get a bootstrap component. The standard way to do this is the following one (this method creates an instance of the class specified in the fractal.provider system property, and uses this instance to get the bootstrap component):

```
 Component boot = Fractal.getBootstrapComponent();
```

We then get the TypeFactory interface provided by this bootstrap component:

```
 TypeFactory tf = (TypeFactory)boot.getFcInterface('type-factory');
```

We can then create the type of the first component, which only provides a A server interface named 'a':

```
// type of the a component
ComponentType aType = tf.createFcType(new InterfaceType[] {
 tf.createFcItfType('a', -'A', false, false, false)
});
```

### 5.1.2. Description of the content

The second step in the definition of a component is the definition of its content. In this implementation, this is done through the ContentDescription class:

```
ContentDescription contentDesc = new ContentDescription(
 AImpl, // signature of the base class
 new Object[] {}, // Object[]
 aNode // location, could also be a virtual node
);
```

### 5.1.3. Description of the controller

Properties relative to the controller can be specified in the ControllerDescription:

```
ControllerDescription controllerDesc = new ControllerDescription(
 -'myName', // name of the component
 Constants.PRIMITIVE // the hierarchical type of the component
 // it could be PRIMITIVE, COMPOSITE, or PARALLEL
);
```

Eventually, the component definition is instantiated using the standard Fractal API. This component can then be manipulated as any other Fractal component.

```
Component component = componentFactory.newFcInstance(
 componentType, // type of the component (defining the client and server interfaces)
 controllerDesc, // implementation-specific description for the controller
 contentDesc // implementation-specific description for the content
);
```

### 5.1.4. From attributes to client interfaces

There are 2 kinds of interfaces for a component: those that offer services, and those that require services. They are named respectively server and client interfaces.

From a Java class, it is fairly natural to identify server interfaces: they (can) correspond to the Java interfaces implemented by the class. In the above example, 'a' is the name of an interface provided by the component, corresponding to the 'A' Java interface.

On the other hand, client interfaces usually correspond to attributes of the class, in the case of a primitive component. If the component defined above requires a service from another component, say the one corresponding to the 'Service' Java interface, the AImpl class should be modified. As we use the **inversion of control** pattern, a BindingController is provided, and a binding operation on the 'requiredService' interface will actually set the value of the 'service' attribute, of type 'Service'.

First, the type of the component is changed:

```
// type of the a component
ComponentType aType = tf.createFcType(new InterfaceType[] {
 tf.createFcItfType('a', -'A', false, false, false),
 tf.createFcItfType('requiredService', -'A', true, false, false)
});
```

The Service interface is the following:

```
package org.objectweb.proactive.examples.components.helloworld;
```

```
public interface Service {
   void print(String msg);
}
```

And the AImpl class is:

```
// The modified AImpl class
public class AImpl implements A, BindingController {
 Service service; // attribute corresponding to a client interface
 public AImpl() {}
 // implementation of the A interface
 public String foo() {
   return service.bar(); // for example
 -}
 // implementation of BindingController
 public Object lookupFc (final String cItf) {
   if (cItf.equals('requiredService')) {
     return service;
   -}
   return null;
 -}
 // implementation of BindingController
 public void bindFc (final String cItf, final Object sItf) {
   if (cItf.equals('requiredService')) {
     service = (Service)sItf;
   -}
 -}
 // implementation of BindingController
 public void unbindFc (final String cItf) {
   if (cItf.equals('requiredService')) {
     service = null;
   -}
 -}
}
```

## 5.2. The HelloWorld example

The mandatory helloworld example (from the Fractal tutorial) shows the different ways of creating a component system (programmatically and using the ADL), and it can easily be implemented using ProActive.

### 5.2.1. Set-up

You can find the code for this example in the package org.objectweb.proactive.examples.components.helloworld of the ProActive distribution.

The code is almost identical to the Fractal tutorial's example [http://fractal.objectweb.org/tutorials/fractal/index.html].

The differences are the following:

- The reference example is provided for level 3.3. implementation, whereas this current implementation is compliant up to level 3.2: templates are not provided. Thus you will have to skip the specific code for templates.
- The newFcInstance method of the GenericFactory interface, used for directly creating components, takes 2 implementation-specific parameters. So you should use the org.objectweb.proactive.component.ControllerDescription and org.objectweb.proactive.component.ContentDescription classes to define ProActive components. (It is possible to use the same parameters than in Julia, but that hinders you from using some functionalities specific to ProActive, such as distributed deployment or definition of the activity).

- Collective interfaces could be implemented the same way than suggested, but using the Fractive.createCollectiveClientInterface method will prove useful with this implementation: you are then able to use the functionalities provided by the typed groups API.
- Components can be distributed
- the ClientImpl provides an empty no-args constructor.

## 5.2.2. Architecture

The helloworld example is a simple client-server application, where the client (c) and the server (s) are components, and they are both contained in the same root component (root).

Another configuration is also possible, where client and server are wrapped around composite components (C and S). The goal was initially to show the interception shortcut mechanism in Julia. In the current ProActive implementation, there are no such shortcuts, as the different components can be distributed, and all invocations are intercepted. The exercise is still of interest, as it involves composite components.

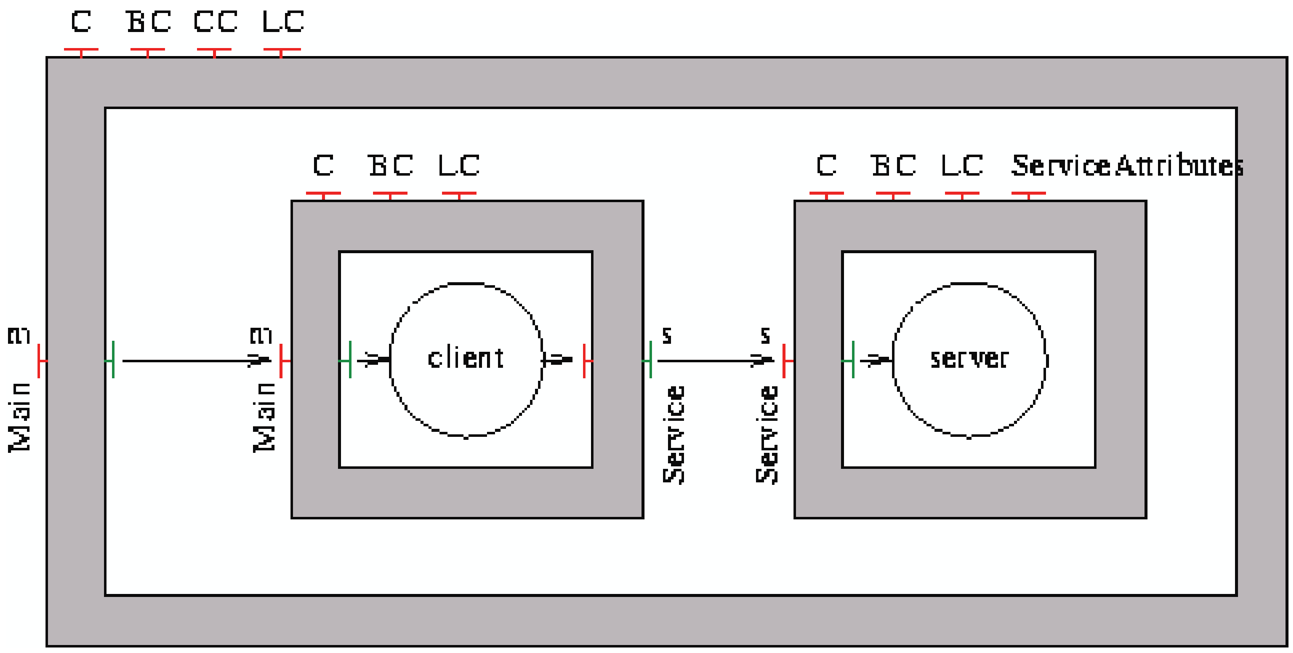


**Figure 5.1. Client and Server wrapped in composite components (C and S)**

## 5.2.3. Distributed deployment

This section is specific to the ProActive implementation, as it uses the deployment framework of this library.

If the application is started with (only) the parameter 'distributed', the ADL used is 'helloworld-distributed-no-wrappers.fractal', where virtualNode of the client and server components are exported as VN1 and VN2. Exported virtual node names from the ADL match those defined in the deployment descriptor 'deployment.xml'.

One can of course customize the deployment descriptor and deploy components onto virtually any computer, provided it is connectable by supported protocols. Supported protocols include LAN, clusters and Grid protocols (see the ProActive manual).

Have a look at the ADL files 'helloworld-distributed-no-wrappers.fractal' and 'helloworld-distributed-wrappers.fractal'. In a nutshell, they say: 'the primitive components of the application (client and server) will run on given exported virtual nodes, whereas the other components (wrappers, root component) will run on the current JVM.

Therefore, we have the two following configurations:

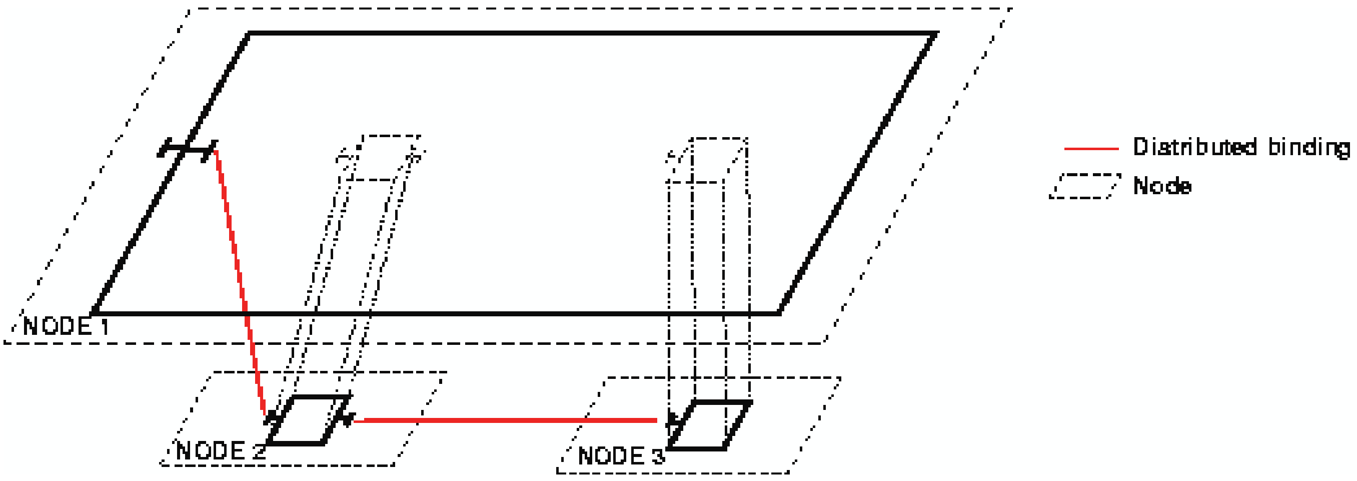


**Figure 5.2. Without wrappers, the primitive components are distributed.**

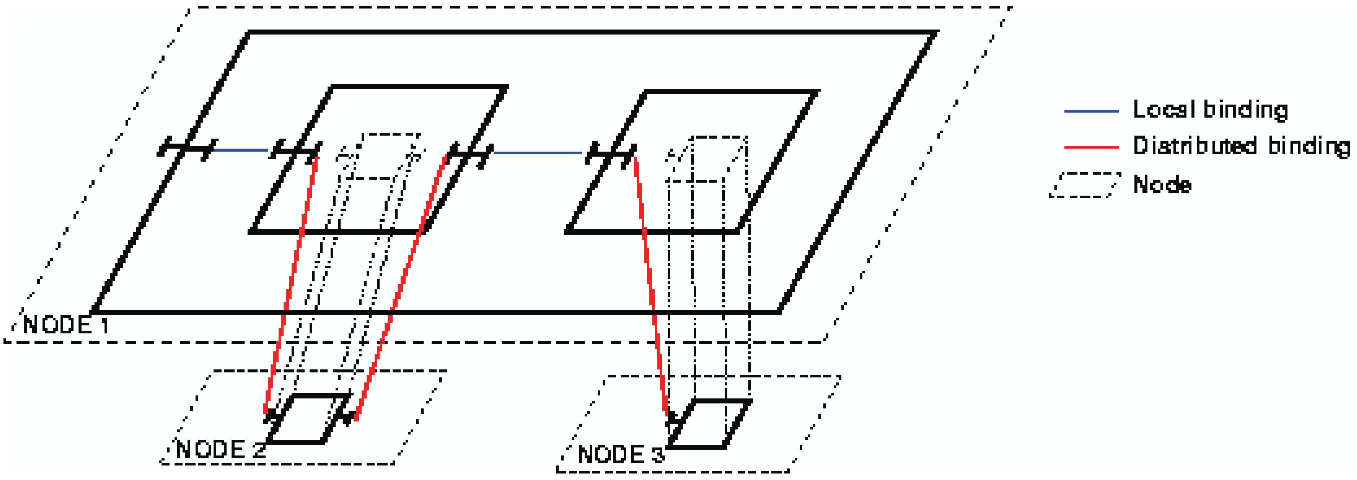


**Figure 5.3. With wrappers, where again, only the primitive components are distributed.**

Currently, bindings are not optimized. For example, in the configuration with wrappers, there is an indirection that can be costly, between the client and the server. We are currently working on optimizations that would allow to shortcut communications, while still allowing coherent dynamic reconfiguration. It is the same idea than in Julia, but we are dealing here with distributed components. It could imply compromises between dynamicity and performance issues.

## 5.2.4. Execution

You can either compile and run the code yourself, or follow the instructions for preparing the examples and use the script helloworld_fractal.sh (or .bat). If you choose the first solution, do not forget to set the fractal.provider system property.

If you run the program with no arguments (i.e. not using the parser, no wrapper composite components, and local deployment) , you should get something like this:

```
---- Fractal Helloworld example ---------------------------------------------
----
---- The expected result is an exception
----

[INFO communication.rmi] Created a new registry on port 6646
[INFO proactive.mop] Generating class -:
  pa.stub.org.objectweb.proactive.core.component.type._StubComposite
[INFO proactive.mop] Generating class -:
  pa.stub.org.objectweb.proactive.core.jmx.util._StubJMXNotificationListener
```

```
11: [INFO proactive.mop] Generating class -:
12:   pa.stub.org.objectweb.proactive.examples.components.helloworld._StubClientImpl
13: [INFO proactive.mop] Generating class -:
14:   pa.stub.org.objectweb.proactive.examples.components.helloworld._StubServerImpl
15:
```

You can see:

- line 6: the creation of a rmi registry
- line 7 to 14: the on-the-fly generation of ProActive stubs (the generation of component functional interfaces is silent)

Then you have (the exception that pops out is actually the expected result, and is intended to show the execution path):

```
 1:Server: print method called
 2:at org.objectweb.proactive.examples.components.helloworld.ServerImpl.print(ServerImpl.java:45)
 3:at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 4:at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 5:at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
 6:at java.lang.reflect.Method.invoke(Method.java:597)
 7:at org.objectweb.proactive.core.mop.MethodCall.execute(MethodCall.java:390)
 8:at
 9:  org.objectweb.proactive.core.component.request.ComponentRequestImpl.
10:    serveInternal(ComponentRequestImpl.java:176)
11:at org.objectweb.proactive.core.body.request.RequestImpl.serve(RequestImpl.java:170)
12:at
13:  org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.
14:    serveInternal(BodyImpl.java:539)
15:at org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.serve(BodyImpl.java:510)
16:at org.objectweb.proactive.core.body.AbstractBody.serve(AbstractBody.java:909)
17:at org.objectweb.proactive.Service.blockingServeOldest(Service.java:175)
18:at org.objectweb.proactive.Service.blockingServeOldest(Service.java:150)
19:at org.objectweb.proactive.Service.fifoServing(Service.java:126)
20:at
21:  org.objectweb.proactive.core.component.body.ComponentActivity$ComponentFIFORunActive.
22:    runActivity(ComponentActivity.java:226)
23:at
24:  org.objectweb.proactive.core.component.body.ComponentActivity.
25:    runActivity(ComponentActivity.java:183)
26:at
27:  org.objectweb.proactive.core.component.body.ComponentActivity.
28:    runActivity(ComponentActivity.java:183)
29:at org.objectweb.proactive.core.body.ActiveBody.run(ActiveBody.java:192)
30:at java.lang.Thread.run(Thread.java:619)
31:Server: begin printing...
32:--------> hello world
33:Server: print done.c
34:
```

What can be seen is very different from the output you would get with the Julia implementation. Here is what happens (from bottom to top of the stack):

- line 30: The active object runs its activity in its own Thread
- line 20-21-22: The default activity is to serve incoming request in a FIFO order
- line 8-9-10: Requests (reified method calls) are encapsulated in ComponentRequestImpl objects
- line 6: A request is served using reflection
- line 2: The method invoked is the print method of an instance of ServerImpl

Now let us have a look at the distributed deployment: execute the program with the parameters 'distributed parser'. You should get something similar to the following:

```
 1: ---- Fractal Helloworld example ---------------------------------------------
 2: ----
 3: ---- The expected result is an exception
 4: ----
 5:
 6: [INFO communication.rmi] Created a new registry on port 6646
 7: [INFO proactive] ************* Reading deployment descriptor:
 8:   file:/home/ProActive/classes/Examples/org/objectweb/proactive/examples/components/
 9:     helloworld/deployment.xml ********************
10: [INFO proactive.deployment] created VirtualNode name=VN1
11: [INFO proactive.deployment] created VirtualNode name=VN2
12: [INFO proactive.deployment] created VirtualNode name=VN3
13: [INFO proactive.mop] Generating class -:
14:   pa.stub.org.objectweb.proactive.core.jmx.util._StubJMXNotificationListener
15: [INFO deployment.log]
16: [INFO deployment.log] 311@saturn.inria.fr --
17:   [INFO proactive.runtime] **** Starting jvm on 138.96.218.113
18: [INFO proactive.events] **** Mapping VirtualNode VN1 with Node:
19:   rmi://138.96.218.113:6646/VN11559562212 done
20: [INFO proactive.mop] Generating class -:
21:   pa.stub.org.objectweb.proactive.examples.components.helloworld._StubClientImpl
22: [INFO deployment.log] 311@saturn.inria.fr --
23:   [INFO communication.rmi] Detected an existing RMI Registry on port 6646
24: [INFO deployment.log]
25: [INFO deployment.log] 97714@saturn.inria.fr --
26:   [INFO proactive.runtime] **** Starting jvm on 138.96.218.113
27: [INFO proactive.events] **** Mapping VirtualNode VN2 with Node:
28:   rmi://138.96.218.113:6646/VN2914088183 done
29: [INFO proactive.mop] Generating class -:
30:   pa.stub.org.objectweb.proactive.examples.components.helloworld._StubServerImpl
31: [INFO deployment.log] 97714@saturn.inria.fr -- [INFO communication.rmi] Detected an existing RMI
32: Registry on port 6646
33: [INFO proactive.mop] Generating class -:
34:   pa.stub.org.objectweb.proactive.core.component.type._StubComposite
35:
```

What is new is:

- line 7-8-9 the parsing of the deployment descriptor
- line 16-17 and 25-26: the creation of 2 virtual machines on the host 'crusoe.inria.fr'
- line 10-11-12: the creation of virtual nodes VN1, VN2 and VN3
- line 18-19 and 27-28: the mapping of virtual nodes VN1 and VN2 to the nodes specified in the deployment descriptor

Then we get the same output than for a local deployment, the activity of active objects is independent from its location.

```
[INFO deployment.log] Server: print method called
[INFO deployment.log] at
  org.objectweb.proactive.examples.components.helloworld.ServerImpl.print(ServerImpl.java:45)
[INFO deployment.log] at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[INFO deployment.log] at
  sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[INFO deployment.log] at
  sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[INFO deployment.log] at java.lang.reflect.Method.invoke(Method.java:597)
[INFO deployment.log] at org.objectweb.proactive.core.mop.MethodCall.execute(MethodCall.java:390)
[INFO deployment.log] at
  org.objectweb.proactive.core.component.request.ComponentRequestImpl.
    serveInternal(ComponentRequestImpl.java:176)
[INFO deployment.log] at
```

```
  org.objectweb.proactive.core.body.request.RequestImpl.serve(RequestImpl.java:170)
[INFO deployment.log] at
  org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.
    serveInternal(BodyImpl.java:539)
[INFO deployment.log] at
  org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.serve(BodyImpl.java:510)
[INFO deployment.log] at
  org.objectweb.proactive.core.body.AbstractBody.serve(AbstractBody.java:909)
[INFO deployment.log] at org.objectweb.proactive.Service.blockingServeOldest(Service.java:175)
[INFO deployment.log] at org.objectweb.proactive.Service.blockingServeOldest(Service.java:150)
[INFO deployment.log] at org.objectweb.proactive.Service.fifoServing(Service.java:126)
[INFO deployment.log] at
  org.objectweb.proactive.core.component.body.ComponentActivity$ComponentFIFORunActive.
    runActivity(ComponentActivity.java:226)
[INFO deployment.log] at
  org.objectweb.proactive.core.component.body.ComponentActivity.
    runActivity(ComponentActivity.java:183)
[INFO deployment.log] at
  org.objectweb.proactive.core.component.body.ComponentActivity.
    runActivity(ComponentActivity.java:183)
[INFO deployment.log] at org.objectweb.proactive.core.body.ActiveBody.run(ActiveBody.java:192)
[INFO deployment.log] at java.lang.Thread.run(Thread.java:619)
[INFO deployment.log] Server: begin printing...
[INFO deployment.log] -->hello world
[INFO deployment.log] Server: print done.
---------------------------------------------------------

```

### 5.2.5. The HelloWorld ADL files

org.objectweb.proactive.examples.components.helloworld.helloworld-distributed-wrappers.fractal

```xml
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name=
"org.objectweb.proactive.examples.components.helloworld.helloworld-distributed-wrappers">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <exportedVirtualNodes>
      <exportedVirtualNode name="VN1">
        <composedFrom>
            <composingVirtualNode component="client" name="client-node"/>
        </composedFrom>
      </exportedVirtualNode>
      <exportedVirtualNode name="VN2">
        <composedFrom>
            <composingVirtualNode component="server" name="server-node"/>
        </composedFrom>
      </exportedVirtualNode>
    </exportedVirtualNodes>
  <component name="client-wrapper" definition=
"org.objectweb.proactive.examples.components.helloworld.ClientType">
    <component name="client" definition=
"org.objectweb.proactive.examples.components.helloworld.ClientImpl"/>
    <binding client="this.r" server="client.r"/>
    <binding client="client.s" server="this.s"/>
    <controller desc="composite"/>
  </component>
  <component name="server-wrapper" definition=
"org.objectweb.proactive.examples.components.helloworld.ServerType">
    <component name="server" definition=
"org.objectweb.proactive.examples.components.helloworld.ServerImpl"/>
    <binding client="this.s" server="server.s"/>
    <controller desc="composite"/>
  </component>
  <binding client="this.r" server="client-wrapper.r"/>
  <binding client="client-wrapper.s" server="server-wrapper.s"/>
</definition>
```

org.objectweb.proactive.examples.components.helloworld.ClientType.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.helloworld.ClientType" extends=
"org.objectweb.proactive.examples.components.helloworld.RootType">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature=
"org.objectweb.proactive.examples.components.helloworld.Service"/>
</definition>
```

org.objectweb.proactive.examples.components.helloworld.ClientImpl.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.helloworld.ClientImpl" extends=
"org.objectweb.proactive.examples.components.helloworld.ClientType">
  <exportedVirtualNodes>
      <exportedVirtualNode name="client-node">
        <composedFrom>
            <composingVirtualNode component="this" name="client-node"/>
        </composedFrom>
      </exportedVirtualNode>
    </exportedVirtualNodes>
    <content class="org.objectweb.proactive.examples.components.helloworld.ClientImpl"/>
    <virtual-node name="client-node" cardinality="single"/>
</definition>
```

org.objectweb.proactive.examples.components.ServerType

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.helloworld.ServerType">
  <interface name="s" role="server" signature=
"org.objectweb.proactive.examples.components.helloworld.Service"/>
</definition>
```

org.objectweb.proactive.examples.components.helloworld.ServerImpl

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
```

```
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.helloworld.ServerImpl" extends=
"org.objectweb.proactive.examples.components.helloworld.ServerType">
  <exportedVirtualNodes>
      <exportedVirtualNode name="server-node">
        <composedFrom>
            <composingVirtualNode component="this" name="server-node"/>
        </composedFrom>
      </exportedVirtualNode>
    </exportedVirtualNodes>
  <content class="org.objectweb.proactive.examples.components.helloworld.ServerImpl"/>
  <attributes signature=
"org.objectweb.proactive.examples.components.helloworld.ServiceAttributes">
    <attribute name="header" value="->"/>
    <attribute name="count" value="1"/>
  </attributes>
  <controller desc="primitive"/>
  <virtual-node name="server-node" cardinality="single"/>
</definition>
```

# Chapter 6. Component perspectives: a support for advanced research

The ProActive/Fractal framework is a functional and flexible implementation of the Fractal API and model. One can configure and deploy a system of distributed components, including Grids. The framework also proposes extensions for collective interactions (gathercast and multicast interfaces), allocation configuration through virtual nodes extensions, and some optimizations.

It is now a mature framework for developing Grid applications, and as such it is a basis for experimenting new research paths.

## 6.1. Dynamic reconfiguration

One of the challenges of Grid computing is to handle changes in the execution environments, which are not predictable in systems composed of large number of distributed components on heterogeneous environments. For this reason, the system needs to be dynamically reconfigurable, and must exhibit autonomic properties.

Simple and deterministic dynamic reconfiguration is a real challenge in systems that contain hierarchical components that feature their own activities and that communicate asynchronously.

A part of the solutions envisioned consist in designing a set of high-level reconfiguration primitives allowing to achieve complex operations, but also to trigger such operations on specific events. This aspects consists in designing a set of such primitives (e.g., replace, add and bind, unbind and remove, duplicate, recursively add, ...) for reconfiguration ensuring more correctness properties than the Fractal ones, and more autonomicity. By providing higher level of primitives, the principal aim is to help the programmer to design safe scenarii. For example a replacement primitive seems safer and easier to verify than the equivalent sequence (stop +unbind+remove+add+bind+start) that would implement it in Fractal. One of the difficulties is that most useful reconfigurations involve changing or augmenting the available behaviors of the system components. During replacement, one can introduce new interfaces, new dependencies between components.

Another issue related to reconfigurations and component life-cycle is the coherency in the component states along reconfigurations. Indeed, suppose for example that two consecutive requests (on the same binding) should necessarily be addressed to the same destination component (for example, the one requests sends additional informations necessary to fulfill the other one). Then, between those two request, no reconfiguration can occur if it involves the binding used for the requests.

As a consequence, it is important designing a way of specifying synchronization between reconfiguration steps and the application, this should be the main interaction between functional and non-functional aspects, and should be studied carefully in order to maintain the "good separation of aspects" that exists in Fractal.

The autonomic computing paradigm is related to this challenge because is consists of building applications out of self-managed components. Components which are self-managed are able to monitor their environment and adapt to it by automatically optimizing and reconfiguring themselves. The resulting systems are autonomous and automatically fulfill the needs of the users, but the complexity of adaptation is hidden to them. Autonomicity of components represents a key asset for large scale distributed computing.

## 6.2. Model-checking

Encapsulation properties, components with configurable activities, and system description in ADL files provide safe basis for model checking of component systems.

For instance:

1. Behavioral information on components can be specified in extended ADL files.
2. Automatas can be generated from behavioral information and structural description.
3. Model checking tools are used to verify the automatas.

The Vercors [http://www-sop.inria.fr/oasis/Vercors/] platform investigates such kinds of scenarii.

Component-based software development (CBSD) has emerged as a response from both industries and academics for dealing with software complexity and reusability. The main idea is to clearly define interfaces between components so that they can be assembled and composed in several contexts. Unfortunately, software engineers often face non-trivial runtime incompatibilities when assembling off-the-shelf components. These arise due to an inadequate (or nonexistent) dynamic specification of the component behaviour. In fact, state-of-the-art implementations of component models such as SOFA , Fractal and CORBA Component Model only consider interface type-compatibility (through Interface Description Languages or IDLs) for binding interfaces. Nonetheless, a sound static compatibility check of bound interfaces can be achieved if behavioural information is added to the components. There are several related works, that either introduce Behavioural IDLs or that describe behaviour of components.

We are building a tool platform for the analysis and verification of safety and security properties of distributed applications. The central component of the platform is a method for generating finite models for distributed applications, from static analysis of source code. We base this generation procedure on the strong semantic features provided by the ProActive library, and we generate compositional models using synchronised labelled transition systems. Various tools for static analysis, model checking, and equivalence checking can then operate on these models. One long term goal of this work is to integrate the various techniques and tools involved in this software platform, so that the platform can be integrated in a development environment, and used by non-specialists. At the same time, the platform must be flexible and open enough to serve as a basis for easy prototyping of new techniques and tools on real Java/ProActive code.

Even if there are many specification languages in the literature, none fits well in the context of distributed components. In the GCM, most difficulties come when specifying the synchronisations. Instead of proving that legacy code is safe, we take a constructive approach. The idea is to specify the system, prove that the specification is correct, and then generate (Java) code skeletons guaranteed to conform to the specification. pNets is left as the underlying formalism that interfaces with model-checkers, and the programmer uses a high-level specification on top of pNets. The language is called **Java Distributed Components** (JDC for short).

## 6.3. Pattern-based deployment

Distributed computational applications are designed by defining a functional or do- main decomposition, and these decompositions often present structural similarities (master-slave, 2D-Grid, pipeline etc.).

In order to facilitate the design of complex systems with large number of entities and recurring similar configurations, we plan to propose a mechanism for defining parameterizable assembly patterns in the Fractal ADL, particularly for systems that contain parameterized numbers of identical components.

## 6.4. Graphical tools

We are developing the VCE (Vercors Component Environnement), that includes graphical editors for the architecture and the behavior of GCM components.

The architecture diagrams traditionally feature hierarchical components, provided and required interfaces (with Java signatures attached), and bindings. But they also distinguish GCM specific concepts, namely functional and non-functional interfaces, content and membrane parts for composite components, multicast and gathercast interfaces. Diagrams are validated against a set of static semantic rules. GCM-ADL files can be produced and read by the editor. The behavior diagrams express external behavior of components. They are based on classical state-machines constructions, with specific constructs for GCM/Proactive, in particular for expressing request queue selection, and multicast/gathercast policies.

# Chapter 7. GCM Components Tutorial

## 7.1. Introduction

This chapter presents a short user guide which explains how to use the ProActive/GCM implementation. The chapter will not explain how to program with components but instead focus on the particularities of the GCM implementation for ProActive.

## 7.2. Creating and using components in a programatic way

Along this short user guide, we will show: how to create primitive and composite components, how to assemble them using Fractal/GCM API and Fractal API files, how to interoperate with components, and then how to describe the deployment of components using deployment descriptor file.

The first step of this user guide explains how to create a single primitive component. Next, we will use an assembly of two primitive components in a composite one.

### 7.2.1. The first component

We want to create a primitive component, called PrimitiveComputer. It exposes one server interface called computer-itf which provides the two following methods: compute and doNothing. To do that, we need to write the two following classes.

```
package org.objectweb.proactive.examples.components.userguide.primitive;

public interface ComputeItf {
  int compute(int a);

  void doNothing();
}
```

```
package org.objectweb.proactive.examples.components.userguide.primitive;

import java.io.Serializable;


public class PrimitiveComputer implements ComputeItf, Serializable {
  public PrimitiveComputer() {
  -}

  public int compute(int a) {
     int result = a * 2;
     System.err.println(" PrimitiveComputer-->compute(" + a + "): -" + result);
     return result;
  -}

  public void doNothing() {
     System.err.println(" PrimitiveComputer-->doNothing()");
  -}
}
```

Now, we will discuss on the different ways to use this component. First, we must create the component with the ProActive/GCM framework. Two kinds of component instantiation are shown. In the first case, we can do all these steps in the application. However, in the second case, we will show how we can use the ADL files to simplify the application and create it in a simpler way.

In order to illustrate these different ways, a new class, Main, containing the possible main method of our application (see the source code below), is written. In this main method, four different methods are called and will be described in the following parts of this document, launchFirstPrimitive, launchWithoutADL, launchWithADL, and finally the last launchAndDeployWithADL. To launch this class, you must put in your classpath all the libraries contained in the lib directory and subdirectories and the ProActive jar. And finally, you must set the three Java properties (fractal.provider, java.security.policy, log4j.configuration) as shown in the command line:

```
java
  --Dfractal.provider=org.objectweb.proactive.core.component.Fractive
  --Djava.security.policy=file:<change_this_part>/
                              ProActive/dist/proactive.java.policy
  --Dlog4j.configuration=file:<change_this_part>/
                             ProActive/dist/proactive-log4j
  org.objectweb.proactive.examples.components.userguide.Main
```

```java
package org.objectweb.proactive.examples.components.userguide;

import java.util.HashMap;
import java.util.Map;

import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.LifeCycleController;
import org.objectweb.fractal.api.factory.GenericFactory;
import org.objectweb.fractal.api.type.ComponentType;
import org.objectweb.fractal.api.type.InterfaceType;
import org.objectweb.fractal.api.type.TypeFactory;
import org.objectweb.fractal.util.Fractal;
import org.objectweb.proactive.api.PADeployment;
import org.objectweb.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ContentDescription;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.factory.ProActiveGenericFactory;
import org.objectweb.proactive.core.descriptor.data.ProActiveDescriptor;
import org.objectweb.proactive.core.descriptor.data.VirtualNode;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.examples.components.userguide.primitive.ComputeItf;
import org.objectweb.proactive.examples.components.userguide.primitive.PrimitiveComputer;
import org.objectweb.proactive.examples.components.userguide.primitive.PrimitiveMaster;


public class Main {
   public static void main(String[] args) {
       //        System.out.println("Launch primitive component example");
       //        Main.launchFirstPrimitive();
       System.out.println("Launch component assembly example");
       Main.launchWithoutADL();

       //        System.out.println("Launch and deploy component assembly example");
       //        Main.launchAndDeployWithoutADL();

       //        System.out.println("Launch component assembly example with ADL");
       //        Main.launchOneWithADL();
       //
       //        System.out.println("Launch and deploy component assembly example with ADL");
       //        Main.launchAndDeployWithADL();

       //System.err.println("The END...");
       //System.exit(0);
   -}
```

If we want to create and call components in a standard Java application, we need to use the GCM API [1]. The method launchFirstPrimitive shows all the steps to create and use our first primitive component. Firstly, define the type of the component. Secondly, create component using a factory. Thirdly, start the component. And finally, retrieve the component's interface and use it as a standard Java object to access our component.

```
  private static void launchFirstPrimitive() {
      try {
          Component boot = Fractal.getBootstrapComponent();
          TypeFactory typeFact = Fractal.getTypeFactory(boot);
          GenericFactory genericFact = Fractal.getGenericFactory(boot);
          Component primitiveComputer = null;

          // type of PrimitiveComputer component
          ComponentType computerType = typeFact.createFcType(new InterfaceType[] { typeFact
                -.createFcItfType("compute-itf", ComputeItf.class.getName(), TypeFactory.SERVER,
                        TypeFactory.MANDATORY, TypeFactory.SINGLE) -});

          // component creation
          primitiveComputer = genericFact.newFcInstance(computerType, new ControllerDescription(
"root",
             Constants.PRIMITIVE), new ContentDescription(PrimitiveComputer.class.getName()));

          // start PrimitiveComputer component
          Fractal.getLifeCycleController(primitiveComputer).startFc();
          ((LifeCycleController) primitiveComputer.getFcInterface("lifecycle-controller"
)).startFc();

          // get the compute-itf interface
          ComputeItf itf = ((ComputeItf) primitiveComputer.getFcInterface("compute-itf"));
          -;
          // call component
          itf.doNothing();
          int result = itf.compute(5);

          System.out.println("Result of computation whith 5 is: -" + result); //display 10
      -} catch (Exception e) {
          e.printStackTrace();
      -}
  -}
```

Uncomment the line calling the launchFirstPrimitive method in the main method, launch it and see below the expected output. The first lines are ProActive log, and at the end, information printed in the component and in the Main class is visible.

### 7.2.2. Define an assembly

Now that we succeeded to create and use a primitive component, we will learn how to use it in a component assembly. First of all, we want use the previous shown primitive component with another primitive component to explain how to define, implement and use client interfaces. Moreover, in order to use composite component, we put the two primitive components in a composite. The Figure 7.1, "Component assembly " shows this assembly.

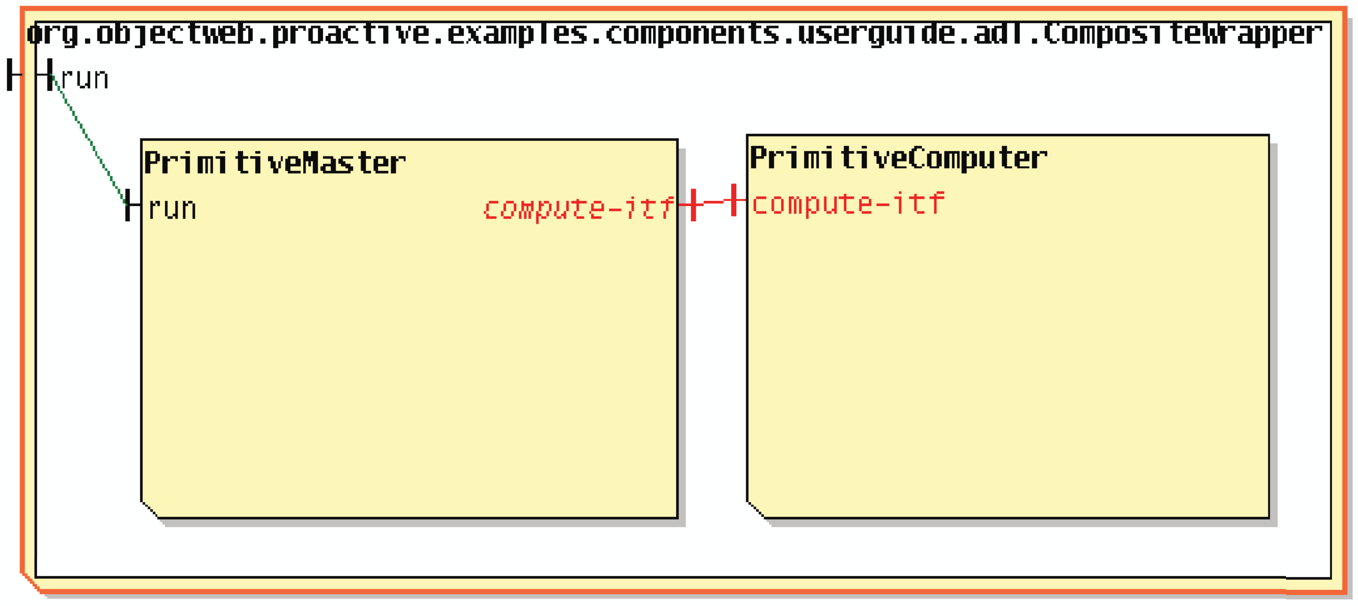

org.objectweb.proactive.examples.components.userguide.adl.CompositeWrapper
run
PrimitiveMaster
run
compute-itf
PrimitiveComputer
compute-itf

**Figure 7.1. Component assembly**

To implement this assembly we need one more class, PrimitiveMaster. This class implements the following Java interfaces: java.lang.Runnable and moreover the BindindController to allow binding on the compute-itf client interface. In the run method we put the call to the PrimitiveComputer component, we don't need to retrieve the compute-itf interface since the assembling it's done in the launchWithoutADL method or in the following part using ADL.

```
package org.objectweb.proactive.examples.components.userguide.primitive;

import java.io.Serializable;

import org.objectweb.fractal.api.NoSuchInterfaceException;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.api.control.IllegalBindingException;
import org.objectweb.fractal.api.control.IllegalLifeCycleException;


public class PrimitiveMaster implements Runnable, Serializable, BindingController {
  private static final String COMPUTER_CLIENT_ITF = "compute-itf";
  private ComputeItf computer;

  public PrimitiveMaster() {
  -}

  public void run() {
      computer.doNothing();
      int result = computer.compute(5);
      System.out.println(" PrimitiveMaster-->run(): -" + "Result of computation whith 5 is: -" +
 result); //display 10
  -}

  //BINDING CONTROLLER implementation
  public void bindFc(String myClientItf, Object serverItf) throws NoSuchInterfaceException,
        IllegalBindingException, IllegalLifeCycleException {
      if (myClientItf.equals(COMPUTER_CLIENT_ITF)) {
          computer = (ComputeItf) serverItf;
```

```
    -}
  -}

  public String[] listFc() {
    return new String[] { COMPUTER_CLIENT_ITF -};
  -}

  public Object lookupFc(String itf) throws NoSuchInterfaceException {
    if (itf.equals(COMPUTER_CLIENT_ITF)) {
      return computer;
    -}
    return null;
  -}

  public void unbindFc(String itf) throws NoSuchInterfaceException, IllegalBindingException,
        IllegalLifeCycleException {
    if (itf.equals(COMPUTER_CLIENT_ITF)) {
      computer = null;
    -}
  -}
}
```

In the launchWithoutADL method, we extend component type definition and component creation parts. And we add one more part, the component assembling. In this part, at first we put the two primitives, PrimitiveComputer and PrimitiveMaster in the composite component. Next, we make the binding between each component interfaces.

```
  private static void launchWithoutADL() {
    try {
        Component boot = Fractal.getBootstrapComponent();
        TypeFactory typeFact = Fractal.getTypeFactory(boot);
        GenericFactory genericFact = Fractal.getGenericFactory(boot);

        // component types: PrimitiveComputer, PrimitiveMaster, CompositeWrapper
        ComponentType computerType = typeFact.createFcType(new InterfaceType[] { typeFact
              -.createFcItfType("compute-itf", ComputeItf.class.getName(), TypeFactory.SERVER,
                    TypeFactory.MANDATORY, TypeFactory.SINGLE) -});
        ComponentType masterType = typeFact.createFcType(new InterfaceType[] {
              typeFact.createFcItfType("run", Runnable.class.getName(), TypeFactory.SERVER,
                    TypeFactory.MANDATORY, TypeFactory.SINGLE),
              typeFact.createFcItfType("compute-itf", ComputeItf.class.getName(),
 TypeFactory.CLIENT,
                    TypeFactory.MANDATORY, TypeFactory.SINGLE) -});
        ComponentType wrapperType = typeFact.createFcType(new InterfaceType[] {
 typeFact.createFcItfType(
              "run", Runnable.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY,
              TypeFactory.SINGLE) -});

        // components creation
        Component primitiveComputer = genericFact.newFcInstance(computerType, new
 ControllerDescription(
            "PrimitiveComputer", Constants.PRIMITIVE), new ContentDescription(PrimitiveComputer.
class
              -.getName()));
        Component primitiveMaster = genericFact.newFcInstance(masterType, new
 ControllerDescription(
            "PrimitiveMaster", Constants.PRIMITIVE), new ContentDescription(PrimitiveMaster.
class
              -.getName()));
        Component compositeWrapper = genericFact.newFcInstance(wrapperType, new
ControllerDescription(
                "CompositeWrapper", Constants.COMPOSITE), null);

        // component assembling
        Fractal.getContentController(compositeWrapper).addFcSubComponent(primitiveComputer);
        Fractal.getContentController(compositeWrapper).addFcSubComponent(primitiveMaster);
        Fractal.getBindingController(compositeWrapper).bindFc("run",
                primitiveMaster.getFcInterface("run"));
        Fractal.getBindingController(primitiveMaster).bindFc("compute-itf",
                primitiveComputer.getFcInterface("compute-itf"));

        // start CompositeWrapper component
        Fractal.getLifeCycleController(compositeWrapper).startFc();

        // get the run interface
        Runnable itf = ((Runnable) compositeWrapper.getFcInterface("run"));

        // call component
        itf.run();
    -} catch (Exception e) {
        e.printStackTrace();
    -}
  -}
```

This way isn't the simplest one to create and use component. There is a lot of code to write, that could introduce mistakes or errors in an assembly. We will show an easier one next.

## 7.3. Create and use components using ADL

We want create the same component directly using ADL capabilities. The source code of the method launchWithADL shows how to use it. Another factory is used, and we can create directly the component without defining at first its type. Utilization of the created component is still the same. You can see that we don't need to define and assemble parts any more. Moreover, we need to create only one component, the other ones are automatically created.

```
  private static void launchWithADL() {
    try {
        Factory f = org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory();
        Map<String, Object> context = new HashMap<String, Object>();

        // component creation
        Component compositeWrapper = (Component) f.newComponent(
                "org.objectweb.proactive.examples.components.userguide.adl.CompositeWrapper",
context);

        // start PrimitiveComputer component
        Fractal.getLifeCycleController(compositeWrapper).startFc();

        // get the run interface
        Runnable itf = ((Runnable) compositeWrapper.getFcInterface("run"));

        // call component
        itf.run();
    -} catch (Exception e) {
        e.printStackTrace();
    -}
  -}
```

ADL allows describing a component assembly through a text file. In our case, we have defined fives files. These files need to be in the classpath of the application, for instance the PrimitiveComputer.fractal file needs to be in the org/objectweb/proactive/examples/components/userguide/adl directory in the classpath. The first one, PrimitiveComputerType.fractal, describes the component type, in particular the interface and the membrane with the tags interface and controller. The second one, PrimitiveComputer.fractal, adds two necessary information: the implementation class with the content tag and a virtual node with the virtual-node tag. These tags are explained in the following section.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition
  name="org.objectweb.proactive.examples.components.userguide.adl.PrimitiveComputerType">
  <interface
    signature="org.objectweb.proactive.examples.components.userguide.primitive.ComputeItf"
    role="server" name="compute-itf" />
  <controller desc="primitive" />
</definition>
```

It is quite the same for the PrimitiveMaster component; just the name and definition class change, and there is one more interface, a client one.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition
  name="org.objectweb.proactive.examples.components.userguide.adl.PrimitiveMasterType">
  <interface
    signature="org.objectweb.proactive.examples.components.userguide.primitive.ComputeItf"
    role="client" name="compute-itf" />
  <interface signature="java.lang.Runnable" role="server" name="run" />
  <controller desc="primitive" />
</definition>
```

And finally, there is the composite one. It defines one interface, and include the two primitive described previously. The binding tag is new; it describes the binding between the interface from composite and inner components.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition
  extends="org.objectweb.proactive.examples.components.userguide.adl.PrimitiveMasterType"
  name="org.objectweb.proactive.examples.components.userguide.adl.PrimitiveMaster">
  <content
    class="org.objectweb.proactive.examples.components.userguide.primitive.PrimitiveMaster" />
  <virtual-node name="primitive-node" cardinality="single" />
</definition>
```

Now, we can run the example; uncomment the line calling the launchWithADL method in the main and then you can see the same output as in the previous section.

## 7.4. Creating, using and deploying components using ADL

To deploy components on a specific virtual node, we need to use ADL files. Just before we saw that the tag virtual-node allows to specify which virtual node to use for a component. The virtual node is defined in a separate file: a deployment descriptor. You can find more information on how to write a deployment descriptor file in the ProActive documentation, chapter 21, XML Deployment Descriptors. The deployment descriptor file used in this example is in the Appendix: deploymentDescriptor.xml.

Furthermore, we need to inform the factory how to use this deployment descriptor; we do this in the launchPrimitiveADLAndDeployment method :

- We create a ProActiveDescriptor object
- We put this object in the context HashMap
- We give this HashMap to the factory

Thus, the factory can retrieve the virtual node defined, and use it as described in the ADL files.

There is another specific point in the end of this method with the deploymentDescriptor.killall(false); call. This method kills all the JVM deployed using the original deployment descriptor file. Before this call, we need to suspend the program since the method calls in GCM are asynchronous, in order to not kill JVM before the end of the component execution.

```
 private static void launchAndDeployWithADL() {
     try {
         // get the component Factory allowing component creation from ADL
         Factory f = org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory();
         Map<String, Object> context = new HashMap<String, Object>();

         // retrieve the deployment descriptor
         ProActiveDescriptor deploymentDescriptor = PADeployment.getProactiveDescriptor(Main.
class
                 -.getResource("deploymentDescriptor.xml").getPath());
         context.put("deployment-descriptor", deploymentDescriptor);
         deploymentDescriptor.activateMappings();

         // component creation
         Component compositeWrapper = (Component) f.newComponent(
                 "org.objectweb.proactive.examples.components.userguide.adl.CompositeWrapper",
 context);

         // start PrimitiveComputer component
         Fractal.getLifeCycleController(compositeWrapper).startFc();

         // get the compute-itf interface
         Runnable itf = ((Runnable) compositeWrapper.getFcInterface("run"));

         // call component
         itf.run();

         Thread.sleep(1000);
          // wait for the end of execution
         // and kill JVM created with the deployment descriptor
         deploymentDescriptor.killall(false);
     -} catch (Exception e) {
         e.printStackTrace();
     -}
 -}
```

Now we can run this example; uncomment the line calling the launchPrimitiveADLAndDeployment method, launch it and see the output. The first lines are ProActive log; it's more verbose than during previous execution because we deploy the two JVMs defined in the deployment descriptor file. After that, you can see information printed from the component and the Main class . And finally, the ProActive log again when the created JVMs are killed.

## 7.5. Component interface Cardinality

Client and server also support multicast and gathercast interface cardinality. The GCM [1] explains which constraints the server and client interfaces must respect.

For multicast interfaces you can specify the parameter dispatching mode thanks to Java annotations available in the org.objectweb.proactive.core.component.type.annotations.multicast package.

## 7.6. Additional examples

Two component applications are included in ProActive the HelloWorld and C3D example.

A Hello World example is provided. It shows the different ways of creating a component system programmatically and using ADL. You can find the code for this example in the package org.objectweb.proactive.examples.components.helloworld of the CFI prototype distribution.

The example code can either be compiled and run manually or using scripts (hello-world_fractal.sh (or .bat) in the scripts/unix/components directory) can be used to launch it. If you choose the first solution, do not forget to set the fractal.provider system property.

The other example, C3D application — a parallel, distributed and collaborative 3D renderer, is in the org.objectweb.proactive.examples.components.c3d package.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3
 http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/deployment.xsd">
  <variables>
    <descriptorVariable name="PROACTIVE_HOME"
      value="/user/cdalmass/home/workspace/ProActiveLatest" -/>
    <!--CHANGE ME!!!! --->
    <descriptorVariable name="JAVA_HOME"
      value="/user/vcave/home/bin/jdk1.6.0_03" -/>
    <!-- -/user/cdalmass/home/pub/local/jdk1.5.0_09 Path of the remote JVM -, CHANGE ME!!!! --->
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="primitive-node" -/>
      <virtualNode name="composite-node" -/>
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="primitive-node">
        <jvmSet>
          <!--      <currentJVM -/>-->
          <vmName value="jvm1" -/>
        </jvmSet>
      </map>
      <map virtualNode="composite-node">
        <jvmSet>
          <!--      <currentJVM -/>-->
          <vmName value="jvm2" -/>
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="jvm1">
        <creation>
          <processReference refid="rshProcess" -/>
        </creation>
      </jvm>
      <jvm name="jvm2">
```

```
        <creation>
          <processReference refid="rshProcess" -/>
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="jvmProcess">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
            <!-- <absolutePath value="${PROACTIVE_HOME}/bin" -/>-->
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Core" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Examples" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Extensions" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Extra" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classesGCMTests" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Tests" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/classes/Utils" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/lib/javassist.jar" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/lib/bouncycastle.jar" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/lib/fractal.jar" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/lib/log4j.jar" -/>
            <absolutePath
              value="${PROACTIVE_HOME}/lib/xercesImpl.jar" -/>
          </classpath>
          <javaPath>
            <absolutePath value="${JAVA_HOME}/bin/java" -/>
          </javaPath>
          <policyFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive.java.policy" -/>
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive-log4j" -/>
          </log4jpropertiesFile>
        </jvmProcess>
      </processDefinition>
      <processDefinition id='rshProcess'>
        <sshProcess
          class='org.objectweb.proactive.core.process.ssh.SSHProcess'
          hostname='nyx.inria.fr'>
          <processReference refid='jvmProcess' -/>
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

# Chapter 8. Annex

## 8.1. The GCM Basics example files

org.objectweb.proactive.examples.components.userguide.starter.Service

```
package org.objectweb.proactive.examples.components.userguide.starter;

public interface Service {
    public void print(String msg);
}
```

org.objectweb.proactive.examples.components.userguide.starter.ServerImpl

```
package org.objectweb.proactive.examples.components.userguide.starter;

public class ServerImpl implements Service {
    public void print(String msg) {
        System.err.println("=> Server: -" + msg);
    -}
}
```

org.objectweb.proactive.examples.components.userguide.starter.Server.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.userguide.starter.Server">
    <interface name="s" role="server" signature=
"org.objectweb.proactive.examples.components.userguide.starter.Service"/>

    <content class="org.objectweb.proactive.examples.components.userguide.starter.ServerImpl"/>

    <controller desc="primitive"/>

    <virtual-node name="VN" cardinality="single"/>
</definition>
```

org.objectweb.proactive.examples.components.userguide.starter.ClientImpl.java

```
package org.objectweb.proactive.examples.components.userguide.starter;

import org.objectweb.fractal.api.control.BindingController;


public class ClientImpl implements Runnable, BindingController {
    private Service service;

    public ClientImpl() {
        // the following instruction was removed, because ProActive requires empty no-args
 constructors
        // otherwise this instruction is executed also at the construction of the stub
        //System.err.println("CLIENT created");
    -}

    public void run() {
        System.err.println("---- Calling service method -----");
        service.print("hello world");
    -}

    public String[] listFc() {
        return new String[] { "s" -};
    -}

    public Object lookupFc(final String cItf) {
        if (cItf.equals("s")) {
            return service;
        -}
        return null;
    -}

    public void bindFc(final String cItf, final Object sItf) {
        if (cItf.equals("s")) {
            service = (Service) sItf;
        -}
    -}

    public void unbindFc(final String cItf) {
        if (cItf.equals("s")) {
            service = null;
        -}
    -}
}
```

org.objectweb.proactive.examples.components.userguide.starter.Client.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" -?>
<!DOCTYPE definition PUBLIC -"-//objectweb.org//DTD Fractal ADL 2.0//EN"
 -"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="org.objectweb.proactive.examples.components.userguide.starter.Client">
  <interface name="m" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature=
"org.objectweb.proactive.examples.components.userguide.starter.Service"/>

  <content class="org.objectweb.proactive.examples.components.userguide.starter.ClientImpl"/>

  <controller desc="primitive"/>

  <virtual-node name="VN" cardinality="single"/>
</definition>
```

org.objectweb.proactive.examples.components.userguide.starter.Main.java

```
package org.objectweb.proactive.examples.components.userguide.starter;

import java.util.HashMap;

import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.util.Fractal;
import org.objectweb.proactive.core.component.adl.FactoryFactory;
import org.objectweb.proactive.core.component.adl.Registry;
import org.objectweb.proactive.core.config.PAProperties;
import org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment;
```

```java
import org.objectweb.proactive.gcmdeployment.GCMApplication;


public class Main {
    public static void main(String[] args) throws Exception {
        PAProperties.FRACTAL_PROVIDER.setValue("org.objectweb.proactive.core.component.Fractive");
        GCMApplication gcma = PAGCMDeployment
                -.loadApplicationDescriptor(Main.class
                        -.getResource(
"/org/objectweb/proactive/examples/components/userguide/starter/applicationDescriptor.xml"));
        gcma.startDeployment();

        Factory factory = FactoryFactory.getFactory();
        HashMap<String, GCMApplication> context = new HashMap<String, GCMApplication>(1);
        context.put("deployment-descriptor", gcma);

        // creates server component
        Component server = (Component) factory.newComponent(
                "org.objectweb.proactive.examples.components.userguide.starter.Server", context);

        // creates client component
        Component client = (Component) factory.newComponent(
                "org.objectweb.proactive.examples.components.userguide.starter.Client", context);

        // bind components
        BindingController bc = ((BindingController) client.getFcInterface("binding-controller"));
        bc.bindFc("s", server.getFcInterface("s"));

        // start components
        Fractal.getLifeCycleController(server).startFc();
        Fractal.getLifeCycleController(client).startFc();

        // launch the application
        ((Runnable) client.getFcInterface("m")).run();

        // stop components
        Fractal.getLifeCycleController(client).stopFc();
        Fractal.getLifeCycleController(server).stopFc();

        Registry.instance().clear();
        gcma.kill();
    -}
}
```

org.objectweb.proactive.examples.components.userguide.starter.deploymentDescriptor.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<GCMDeployment xmlns="urn:gcm:deployment:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:deployment:1.0
 http://proactive.inria.fr/schemas/gcm/1.0/ExtensionSchemas.xsd  -">

  <environment>
    <javaPropertyVariable name="user.home"/>
  </environment>

  <resources>
    <host refid="localhost"/>
  </resources>

  <infrastructure>
    <hosts>
      <host id="localhost" os="unix" hostCapacity="1" vmCapacity="2">
        <homeDirectory base="root" relpath="${user.home}" -/>
      </host>
    </hosts>
  </infrastructure>

</GCMDeployment>
```

org.objectweb.proactive.examples.components.userguide.starter.applicationDescriptor.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:gcm:application:1.0
 http://proactive.inria.fr/schemas/gcm/1.0/ApplicationDescriptorSchema.xsd">

 <environment>
  <javaPropertyVariable name="proactive.home" -/>
  <javaPropertyVariable name="java.home" -/>
  <javaPropertyVariable name="user.dir" -/>
 </environment>


 <application>
  <proactive base="root" relpath="${proactive.home}">
   <configuration>
     <java base="root" relpath="${java.home}/bin/java"/>
    <proactiveClasspath type="append">
     <pathElement base="proactive" relpath="classes/Examples"/>
     <pathElement base="proactive" relpath="dist/lib/clover.jar"/>
    </proactiveClasspath>
   </configuration>
   <virtualNode id="VN">
    <nodeProvider refid="main-VN" -/>
   </virtualNode>
  </proactive>
 </application>

 <resources>
  <nodeProvider id="main-VN">
   <file path="deploymentDescriptor.xml" -/>
  </nodeProvider>
 </resources>
</GCMApplication>
```