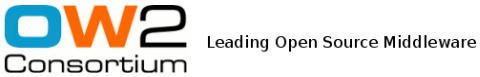
An Open Source Solution for Enterprise Grids & Clouds

ProActive Programming

Components

Version 2014-02-17

ActiveEon Company, in collaboration with INRIA



Generated on 2014-02-17

ProActive Programming v2014-02-17 Documentation

Legal Notice

Contact: proactive@ow2.org or contact@activeeon.com

Mailing List

proactive@ow2.org

Mailing List Archive

http://www.objectweb.org/wws/arc/proactive

Bug-Tracking System

http://bugs.activeeon.com/browse/PROACTIVE

Contributors and Contact Information

Team Leader

Denis Caromel
INRIA 2004, Route des Lucioles, BP 93
06902 Sophia Antipolis Cedex
France
phone: +33 492 387 631
fax: +33 492 387 971
e-mail: Denis.Caromel@inria.fr

Contributors from OASIS Team	Contributors from ActiveEon Company
Brian Amedro	Vladimir Bodnartchouk
Francoise Baude	Arnaud Contes
Francesco Bongiovanni	Cédric Dalmasso
Florin-Alexandru Bratu	Christian Delbé
Viet Dung Doan	Arnaud Gastinel
Yu Feng	Jean-Michel Guillaume
Imen Filali	Olivier Helin
Fabrice Fontenoy	Clément Mathieu
Ludovic Henrio	Maxime Menant
Fabrice Huet	Emil Salageanu
Elaine Isnard	Jean-Luc Scheefer
Vasile Jureschi	Mathieu Schnoor
Muhammad Khan	
Virginie Legrand Contes	
Eric Madelaine	
Elton Mathias	
Paul Naoumenko	
Laurent Pellegrino	
Guilherme Peretti-Pezzi	
Franca Perrina	
Marcela Rivera	
Christian Ruz	
Bastien Sauvan	
Oleg Smirnov	
Marc Valdener	
Fabien Viale	

Former Important Contributors

Laurent Baduel (Group Communications)
Vincent Cave (Legacy Wrapping)
Alexandre di Costanzo (P2P, B&B)
Abhijeet Gaikwad (Option Pricing)
Mario Leyton (Skeleton)
Matthieu Morel (Initial Component Work)
Romain Quilici
Germain Sigety (Scheduling)
Julien Vayssiere (MOP, Active Objects)

Table of Contents

List of Figures

Preface

In order to make the ProActive Programming documentation easier to read, it has been split into four manuals:

- **ProActive Get Started Manual** - This manual contains an overview of ProActive Programming showing different examples and explaining how to install the middleware. It also includes a tutorial teaching how to use it. This manual should be the first one to be read for beginners.
- **ProActive Reference Manual** - This manual is the main manual where the concepts of ProActive are described. Information on configuration and deployment are described in that manual. It also includes guides for high-level APIs usage such as Master-Worker, SMPD APIs.
- **ProActive Advanced Features Manual** - This manual describes some advanced features like Fault-Tolerance, ProActive Compile-Time Annotations or Web Services Exportation. In Addition, it gives information on some other high-level APIs such as Monte-Carlo or Branch and Bound APIs. Finally, it helps advanced user to extend ProActive.
- **ProActive Components Manual** - ProActive defines a component model called ProActive/GCM suitable to support the development of efficient grid applications. This manual therefore contains all necessary information to be able to understand and use this component model. This model is really linked to ProActive so a ProActive/GCM user may have to refer to the ProActive manuals form time to time.

These manuals should be read in the order defined above. However, this is not essential as these documentations are linked together. Besides, if some links seems to be dead in one of these manuals, make sure that all of them had been built previously (multiple html version). So as to build all the manuals at once, go to your ProActive compile/ directory and type build[.bat] doc.ProActive.manualHtml. This builds all manuals in all formats. You may also need the javadoc documentation since these manuals sometime refer to it. To build all the javadoc documentations, type build[.bat] doc.ProActive.javadoc.complete doc.ProActive.javadoc.published inside the compile/ directory. If you just want to build one of these manuals in a specific format, type build[.bat] to see all the possible targets and chose the one you are interested in.

Part I. Guided Tour and Tutorial

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 [1]. The GCM was defined by the CoreGRID NoE project [2] and standardized by ETSI [3]. The complete GCM specification is available here [4] and the GCM standards are available on the ETSI web site:

- GCM Fractal Management API [5]
- GCM Fractal Architecture Description Language [6]

This part documents the ProActive/GCM reference implementation developed by the GridCOMP European project [7].

1.2. Programming with components: the Fractal component model

Fractal defines a general conceptual model, along with an Application Programming 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 either be primitives or composites. A composite is a component containing 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 or membrane). 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:

[1] http://fractal.objectweb.org
[2] http://www.coregrid.net/
[3] http://www.etsi.org/WebSite/homepage.aspx
[4] http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf
[5] http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=28857
[6] http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=28856
[7] http://gridcomp.ercim.org/

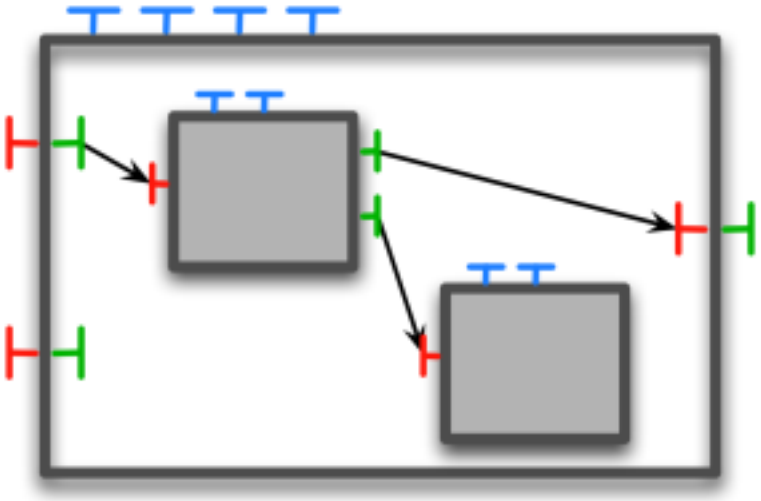


Figure 1.1. A system of Fractal components

Server interfaces, client interfaces and controllers are respectively represented in red, green and blue.

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 here [8].

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:

[8] http://fractal.objectweb.org/specification/fractal-specification.pdf

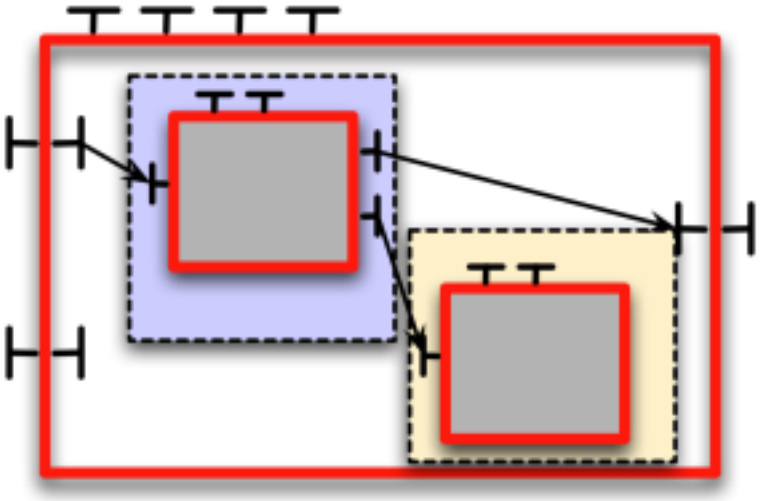


Figure 1.2. A system of distributed ProActive/GCM components (blue, yellow and white represent distinct locations)

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 system 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.

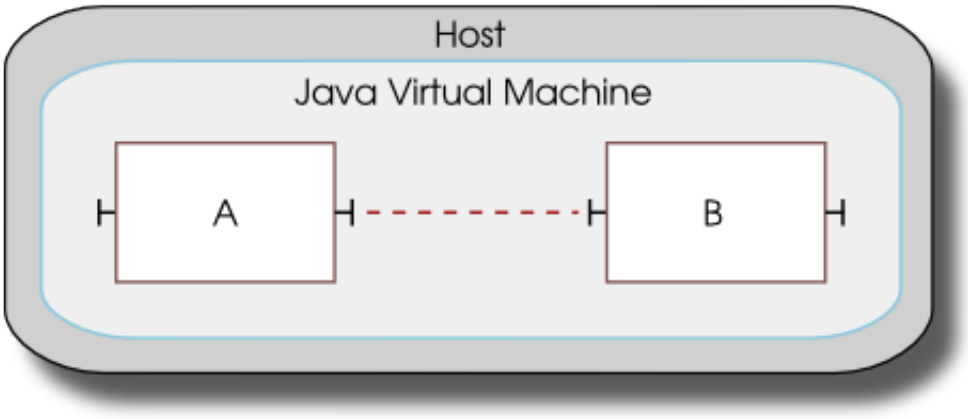
To sum it up, ProActive/GCM mainly provides:

- 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:

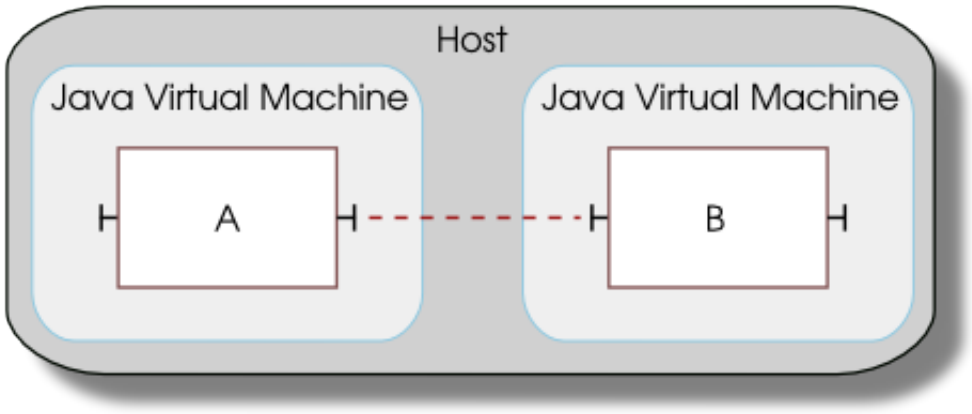
```
<host id="localhost" os="unix" hostCapacity="1"
vmCapacity="2">
```

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, *Appendix*.

Chapter 2. ProActive Example Applications Using Components

2.1. C3D - Active Objects to Components

The standard C3D example has been taken as a basis and component wrappers have been created. This is an example of an application that is refactored to fit the component paradigm. This way, one can see what is needed to transform an application into component-oriented code.

You may find the code in the ProActive/src/Examples/org/objectweb/proactive/examples/components/c3d directory of the proactive source.

2.1.1. Refactoring C3D with components

Add wrappers around the original object classes (C3D*) and instead of linking the classes together by setting fields through the initial methods, do that in the binding methods. In other words, we have to spot exactly where C3DRenderingEngine, C3DUser and C3DDispatcher are used by a class other than itself, and turn these references into component bindings. Obviously, we also have to expose the interfaces that we are going to use, that is to say, the Dispatcher, Engine and User interfaces that have to be implemented.

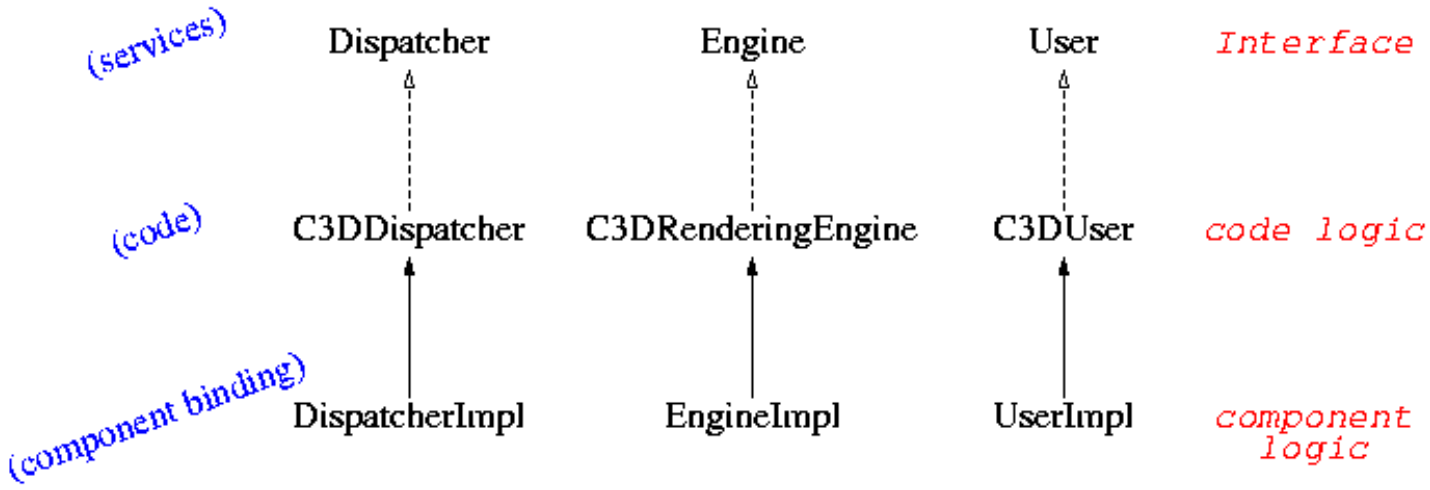


Figure 2.1. Informal description of the C3D Components hierarchy

First of all, have a look at the doc on C3D to remember how this application is written, in Chapter 5.1. C3D: A distributed 3D renderer[1]. The most important thing to keep in mind is the class diagram, showing C3DUser, C3DDispatcher and C3DRederingEngine. We decided that the only objects that worth wrapping in components were those three, remaining objects being too small to get worried about them.

2.1.2. Creating the interfaces

What we need to do is to extract the object interfaces, i.e. to find which methods are going to be called on the components. This means find out what methods are called from outside the active object. You can do that by searching in the classes where the calls are made on active objects. For this, **you have to know in detail which classes are going to be turned into component** . If you have a code base which closely follows object oriented programming rules, the interfaces are already there. Indeed, when a class is written, it should

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../GetStarted/multiple_html/ExampleApplications.html#C3D_example

always go with one or more interfaces, which present to the world what the class abilities are. In C3D (Active Object version), these interfaces already exist: they are called User , Engine and Dispatcher .

Note

Tricky part: whatever way you look at components, you will have to modify the initial code if these interfaces were not created at first go. You have to replace all the class references by their interface, when you use them in other files. For example, if we had not already used interfaces in the C3D object code, we would have had to replace all occurrences of C3DDispatcher by occurrences of Dispatcher.

Why do we have to do that, replacing classes by interfaces? That is due to the way components work. When the components are going to be bound, you are not binding the classes themselves (i.e. the container which performs operations), but proxies to the interfaces presenting the behaviour available. And these proxies implement the interfaces, and do not extend the classes. What is highlighted here is that components enforce good code design by separating behaviours.

2.1.3. Creating the Component Wrappers

You now have to create a class that embraces the previous active objects, and which is a component representing the same functionality. How do you do that? Pretty simple. All you need to do is to extend the active object class, and add to it the non-functional interfaces which go with the component. You have the binding interfaces to create, which basically say how to put together two components, tell who is already attached, and how to separate them. These are the lookupFc , listFc , bindFc and unbindFc methods.

This has been done in the *Impl files. Let's consider, for example, the UserImpl class. What you have here are those component methods. Be even more careful with this bindFc method. In fact, it really binds the protected Dispatcher variable c3ddispatcher as shown hereafter:

```
/** Binds to this UserImpl component the dispatcher which should be used. */
public void bindFc(final String interfaceName, final Object serverInterface) {
   if (interfaceName.equals("user2dispatcher")) {
      c3ddispatcher = (org.objectweb.proactive.examples.c3d.Dispatcher) serverInterface;

      // Registering back to the dispatcher is done in the go() method
   }
}
```

This way, the C3DUser code can now use this variable as if it was addressing the real active object. To be accurate, we have to point out that you are going through proxies before reaching the component, then the active object. This is hidden by the ProActive layer. All you should know is that you are addressing a Dispatcher. The findDispatcher method has been overridden because component lookup does not work like standard Active Object lookup.

2.1.4. Component lookup and registration

When running the User Component alone, you are prompted for an address on which to lookup a Dispatcher Component. Then, the two components are bound through a lookup mechanism. This is very simple to use. Here is the code to do that:

The component Registration

```
Fractive.registerByName(Fractive.getComponentRepresentativeOnThis(), "Dispatcher");
```

The Component lookup

```
/* COMPONENT_ALIAS = "Dispatcher" */
PAComponentRepresentative a;
a = Fractive.lookup(URIBuilder.buildURI(hostName, COMPONENT_ALIAS, protocol, portNumber)
```

```
        .toString());
this.c3dDispatcher = (Dispatcher) a.getFcInterface("user2dispatcher");
```

For the registration, you only need a reference on the component you want to register and a name to give to this component.

The Fractive.lookup method uses a URL to find the host which holds the component. This URL contains the machine name of the host, the communication protocol and the port number, but also the lookup name under which the desired component has been registered under, here "Dispatcher". The last operation consists only in retrieving the correct interface to which to connect to. If the interface is not known at compile-time, it can be discovered at run-time with the getFcInterfaces() method, which lists all the available interfaces.

2.1.5. How to run components C3D

There is only one access point for this example in the scripts directory:

```
examples/c3d$ ./c3d_component.sh
--- Fractal C3D example ----------------------------------------------
Parameters : descriptor_file [fractal_ADL_file]
        The first file describes your deployment of computing nodes.
                You may want to try ../../../descriptors/components/C3D_all.xml
        The second file describes your components layout.
                Default is org.objectweb.proactive.examples.components.c3d.adl.userAndComposite
----------------------------------------------------------
```

There are two ways to start the components C3D. If you only want to start the composite (Dispatcher + Renderer), type:

```
examples/c3d$ ./c3d_component.sh \
../../../descriptors/components/C3D_all.xml \
org.objectweb.proactive.examples.components.c3d.adl.compositeOfDispRend
```

If you want to start only a User, you will be asked for the address of a Dispatcher to which to connect to:

```
examples/c3d$ ./c3d_component.sh \
../../../descriptors/components/C3D_all.xml \
org.objectweb.proactive.examples.components.c3d.adl.UserImpl
```

2.2. Conclusion

These are some examples amongst all the ProActive examples present in the ProActive distribution. To see a full list of examples, please refer to the application[2] web page.

[2] http://proactive.inria.fr/index.php?page=applications

Chapter 3. GCM Components Tutorial

3.1. Introduction

This chapter presents a short user guide which explains how to use the ProActive/GCM implementation. It is composed of two parts: the first one shows how to use components with ADL files and the second one describes how to use them in a programmatic way. If you want more information about GCM Components, please refer to Part II, "Programming With Components".

As in the active object tutorial[1], ProActive provides you with an empty tutorial which enables you to fill in code sources and test them easily. To build the tutorial directory, go to the ProActive/compile directory and type:

```
build tutorials
```

This command will create a new directory in the ProActive home directory, called tutorials. This directory will be composed of four sub-directories:

- src - source directory. You will find in this directory all sources to fill in.
- compile - compilation directory. In this directory, you will be able to compile your code. Type build to know all the available targets. Normally, you will see one target per tutorial example. Targets will be described after each example throughout this chapter.
- scripts - launch scripts directory. You will find in this directory all scripts to launch your compiled code. Scripts will be described after each example throughout this chapter.
- dist - library directory. You will find in this directory all libraries needed to compile your code. Normally, you will not have to deal with this directory.

3.2. Create and use components using ADL

3.2.1. Introduction

This first part of the tutorial shows you how to use ADL files in order to construct GCM components. ADL stands for Architecture Description Language and it allows to describe a component assembly thought an XML file. This part is composed of five exercises:

- **Starter** - this exercise make you create a simple primitive component called Slave
- **Composite** - this exercise make you create a composite component composed of a Master and a Slave component.
- **Interfaces** - this exercise make you modify the previous Slave component to have it implement two server interfaces.
- **Multicast** - this exercise make you modify the previous Master component so as to have its client interface be a multicast interface bound to the server interface of two different Slave components.
- **Deployment** - this exercise make you deploy your component on different nodes located in your local host.

3.2.2. Starter tutorial

3.2.2.1. Tutorial description

This tutorial explains how to create a simple component named Slave using ADL files. This component looks like that:

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../GetStarted/multiple_html/ActiveObjectTutorial.html

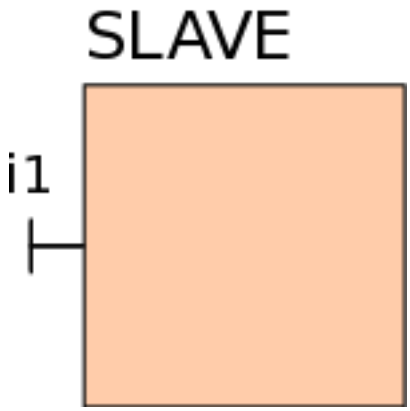


Figure 3.1. Slave component

The interface used for this example is the following one:

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

import java.util.List;


/**
 * @author The ProActive Team
 */
public interface Itf1 {
    void compute(List<String> arg);
}
```

This interface defines a compute() method which is implemented in the following SlaveImpl class:

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

import java.util.List;

import org.objectweb.proactive.api.PAActiveObject;


/**
 * @author The ProActive Team
 */
public class SlaveImpl implements Itf1 {

    public void compute(List<String> arg) {
        String str = "\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this + "\n";
        str += "arg: ";
        for (int i = 0; i < arg.size(); i++) {
            str += arg.get(i);
            if (i + 1 < arg.size()) {
                str += " - ";
            }
        }
        System.err.println(str);
```

```
    }
}
```

In order to define the component, we need an ADL file which describes all the interfaces, the component content class as well as potential binding between interfaces. In that case, the file is very simple:

```
<!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.userguide.components.adl.starter.adl.Slave">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.starter.Itf1" role="server" name="i1"
>

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

   <controller desc="primitive"/>
</definition>
```

With the interface tag, we can define components interfaces. In our case, there is only one interface whose signature is defined in org.objectweb.proactive.examples.userguide.components.adl.starter.Itf1, which is a server interface and whose name is i1. As for the content tag, it enables to define the component contents. Eventually, the controller tag indicates whether the component is a primitive component or a composite one (component composed of several components).

3.2.2.2. Proposed work

Now that we have created our Slave component, it still remain to instantiate it and to use it. That is our proposed work for this first exercise.

Here is the main class that we want you to fill in:

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

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.proactive.core.component.adl.FactoryFactory;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {

      // TODO: Get the Factory
      // TODO: Create Component
      // TODO: Start Component
```

```
        // TODO: Get the interface i1
        // TODO: Call the compute method
        // TODO: Stop Component

        System.exit(0);
    }
}
```

Thus, we propose you to:

1. Get the factory using the org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory() method.
2. Create the Slave component using the newComponent method of the factory previously retrieved. You will need to put in parameters the name of your component definition (written in the ADL file) as well as a Map<String, Object> defining a context.
3. Start the component using the GCM.getGCMLifeCycleController() method to get the life cycle controller of your component and then, using the startFc() on this controller, get your component started.
4. Get the i1 interface using the getFcInterface() method on your component.
5. Call the compute() method with the parameters you want
6. Stop the component using a way quite similar to the one you proceed for starting it.

3.2.2.3. Test your work

To build your work, go to the compile directory into the tutorials directory (not the one located into the ProActive directory) and type:

```
build components.adl.starter
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the adl-starter.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node494021116Slave:
 org.objectweb.proactive.examples.userguide.components.adl.starter.SlaveImpl@15fa713
arg: hello - bye
```

3.2.2.4. Solutions and full code

Here is how you should have filled in the main() method:

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

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.proactive.core.component.adl.FactoryFactory;


/**
 * @author The ProActive Team
 */
public class Main {
```

```
  public static void main(String[] args) throws Exception {

    // TODO: Get the Factory
    Factory factory = FactoryFactory.getFactory();
    // TODO: Create Component
    Map<String, Object> context = new HashMap<String, Object>();
    Component slave = (Component) factory.newComponent(
          "org.objectweb.proactive.examples.userguide.components.adl.starter.adl.Slave", context);
    // TODO: Start Component
    GCM.getGCMLifeCycleController(slave).startFc();
    // TODO: Get the interface i1
    Itf1 itf1 = (Itf1) slave.getFcInterface("i1");
    // TODO: Call the compute method
    List<String> arg = new ArrayList<String>();
    arg.add("hello");
    arg.add("world");
    itf1.compute(arg);
    // TODO: Stop Component
    GCM.getGCMLifeCycleController(slave).stopFc();

    System.exit(0);
  }
}
```

3.2.3. Composite tutorial

3.2.3.1. Tutorial description

This tutorial shows how to create a composite component composed of a Master and a Slave component. This tutorials aims at making you manipulate ADL files. Java sources are filled in but they will be explain to you however. These explanations will be useful for the next tutorials.

This composite can be represented as follows:

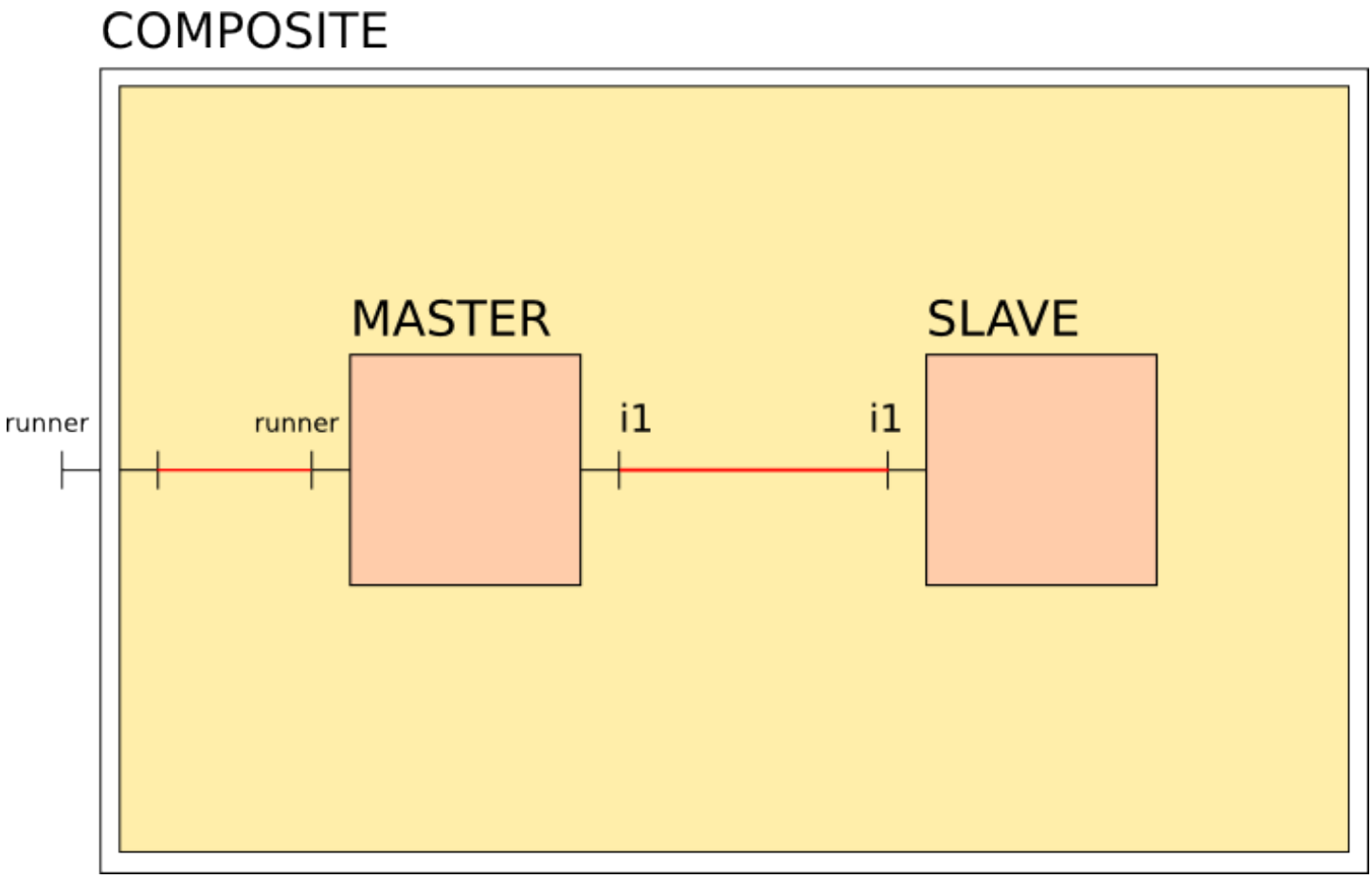


Figure 3.2. Composite component

The Slave component is exactly the same as in the previous tutorial. It is defined by the Itf1 interface implemented by the SlaveImpl class and the corresponding ADL file is Slave.fractal.

Concerning the Master component, it is defined by the following Runner interface:

```
package org.objectweb.proactive.examples.userguide.components.adl.composite;

import java.util.List;


/**
 * @author The ProActive Team
 */
public interface Runner {
    public void run(List<String> arg);
}
```

which is implemented by the MasterImpl class:

```
package org.objectweb.proactive.examples.userguide.components.adl.composite;

import java.util.List;
```

```
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;


/**
 * @author The ProActive Team
 */
public class MasterImpl implements Runner, BindingController {
    public static String ITF_CLIENT = "i1";
    private Itf1 i1;

    public void run(List<String> arg) {
        i1.compute(arg);
    }

    public void bindFc(String clientItfName, Object serverItf) throws NoSuchInterfaceException,
            IllegalBindingException, IllegalLifeCycleException {
        if (ITF_CLIENT.equals(clientItfName)) {
            i1 = (Itf1) serverItf;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

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

    public Object lookupFc(String clientItfName) throws NoSuchInterfaceException {
        if (ITF_CLIENT.equals(clientItfName)) {
            return i1;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

    public void unbindFc(String clientItfName) throws NoSuchInterfaceException, IllegalBindingException,
            IllegalLifeCycleException {
        if (ITF_CLIENT.equals(clientItfName)) {
            i1 = null;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

}
```

This class contains an Itf1 member representing the client interface which is used in the BindingController method. These methods are:

- bindFc() - used to bind interfaces
- listFc() - used to list interfaces
- lookupFc() - used to get one of the available interfaces using its name

- unbindFc() - used to unbind interfaces

The ADL file corresponding to this component is the Master.fractal file which is almost empty since it will be one of your work.

The only things that remain to do now so as to be able to use our composite component are first, to define that composite with an ADL file and then to write our main method to use it. The writing of the composite ADL file is the second work you have to do and the main method, which is quite similar to the one you have written in the previous tutorial, is exposed hereafter.

```
package org.objectweb.proactive.examples.userguide.components.adl.composite;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.proactive.core.component.adl.FactoryFactory;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {
      Factory factory = FactoryFactory.getFactory();

      Map<String, Object> context = new HashMap<String, Object>();
      Component composite = (Component) factory.newComponent(
            "org.objectweb.proactive.examples.userguide.components.adl.composite.adl.Composite", context);

      GCM.getGCMLifeCycleController(composite).startFc();

      Runner runner = (Runner) composite.getFcInterface("runner");
      List<String> arg = new ArrayList<String>();
      arg.add("hello");
      arg.add("world");
      runner.run(arg);

      GCM.getGCMLifeCycleController(composite).stopFc();

      System.exit(0);
   }
}
```

You can notice that the only thing that changed is the first argument of the newComponent() method which is now the name of the ADL file representing the composite component.

3.2.3.2. Proposed work

Here are the two ADL files that you have to fill in:

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

```
<!-- TODO: Write the Master Component definition -->
<!-- This Component has 2 interfaces: -->
<!-- Server: org.objectweb.proactive.examples.userguide.components.adl.composite.Runner -->
<!-- Client: org.objectweb.proactive.examples.userguide.components.adl.composite.Itf1 -->
```

```
<!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.userguide.components.adl.composite.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.composite.Runner" role="server" na
>

  <!-- TODO: Add the Master Component -->

 <component name="Slave" definition="org.objectweb.proactive.examples.userguide.components.adl.composite.adl.Slav
>

  <!-- TODO: Do the bindings -->

  <!-- TODO: Indicates that this component is a composite component -->
</definition>
```

We propose you to:

1. Write the definition of the Master ADL file. Hint: draw your inspiration from the Slave ADL file.
2. In the Composite ADL file, define the Master component
3. Write the bindings. There are two bindings: one for the runner interfaces and another one for the i1 interfaces. Hint: use the <binding client="..." server="..."> tag.
4. Add the controller tag to indicate the type of component

3.2.3.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.adl.composite
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the adl-composite.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node82553583Slave:
 org.objectweb.proactive.examples.userguide.components.adl.composite.SlaveImpl@50078e
arg: hello - world
```

3.2.3.4. Solutions and full code

Here is how you should have filled in the two ADL files:

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

<!-- TODO: Write the Master Component definition -->
<!-- This Component has 2 interfaces: -->
```

```
<!-- Server: org.objectweb.proactive.examples.userguide.components.adl.composite.Runner -->
<!-- Client: org.objectweb.proactive.examples.userguide.components.adl.composite.Itf1 -->
<definition name="org.objectweb.proactive.examples.userguide.components.adl.composite.adl.Master">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.composite.Runner" role="server" na
>

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.composite.Itf1" role="client" name=
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.composite.MasterImpl"/>

  <controller desc="primitive"/>
</definition>
```

```
<!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.userguide.components.adl.composite.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.composite.Runner" role="server" na
>

  <!-- TODO: Add the Master Component -->

 <component name="Master" definition="org.objectweb.proactive.examples.userguide.components.adl.composite.adl.Ma
>

 <component name="Slave" definition="org.objectweb.proactive.examples.userguide.components.adl.composite.adl.Slav
>

  <!-- TODO: Do the bindings -->
  <binding client="this.runner" server="Master.runner"/>
  <binding client="Master.i1" server="Slave.i1"/>

  <!-- TODO: Indicates that this component is a composite component -->
  <controller desc="composite"/>
</definition>
```

3.2.4. Interface tutorial

3.2.4.1. Tutorial description

In this tutorial, we will add a new server interface to the Slave component and bind it to a new client interface of the master component. Our composite component will therefore look like as follows:

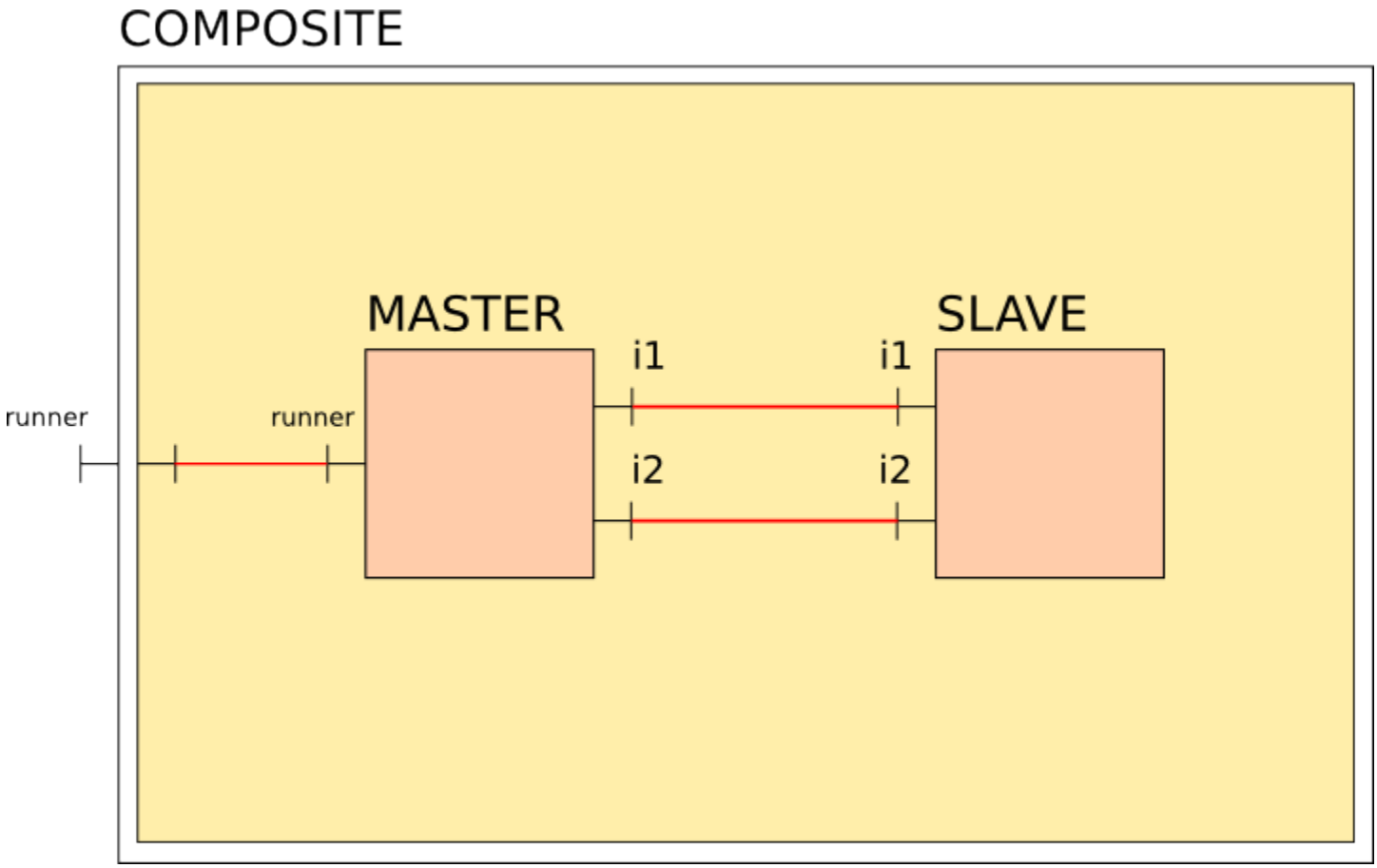


Figure 3.3. Composite component with the new interface

We introduce a second interface named Itf2 which source code is the following one:

```
package org.objectweb.proactive.examples.userguide.components.adl.interfaces;

/**
 * @author The ProActive Team
 */
public interface Itf2 {
    void doNothing();
}
```

The aim of this tutorial is to make you modify all the necessary files to add this new interface.

3.2.4.2. Proposed work

The first thing to do when adding a new interface is to implement it. In our case, it is SlaveImpl class that implements it:

```
package org.objectweb.proactive.examples.userguide.components.adl.interfaces;

import java.util.List;

import org.objectweb.proactive.api.PAActiveObject;
```

```
/**
 * @author The ProActive Team
 */
public class SlaveImpl implements Itf1
//TODO: Add the new interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2
{

   public void compute(List<String> arg) {
      String str = "\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this + "\n";
      str += "arg: ";
      for (int i = 0; i < arg.size(); i++) {
         str += arg.get(i);
         if (i + 1 < arg.size()) {
            str += " - ";
         }
      }
      System.err.println(str);
   }

   public void doNothing() {
      System.err.println("\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this +
         "\nI do nothing");
   }
}
```

Then, you have to modify the ADL file of the Slave component to add this new interface:

```
<!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.userguide.components.adl.interfaces.adl.Slave">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf1" role="server" name=
>
  <!-- TODO: Add the server interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2 -->

  <content class="org.objectweb.proactive.examples.userguide.components.adl.interfaces.SlaveImpl"/>

  <controller desc="primitive"/>
</definition>
```

Now, you have to do the same thing for the Master component, that is, you have to add a new interface. However, in that case, you have to add a client interface and thus, you have to modify the BindingController methods:

```
package org.objectweb.proactive.examples.userguide.components.adl.interfaces;

import java.util.List;

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;
```

```
/**
 * @author The ProActive Team
 */
public class MasterImpl implements Runner, BindingController {
   public static String ITF_CLIENT_1 = "i1";
   public static String ITF_CLIENT_2 = "i2";
   private Itf1 i1;
   private Itf2 i2;

   public void run(List<String> arg) {
      i1.compute(arg);
      i2.doNothing();
   }

   public void bindFc(String clientItfName, Object serverItf) throws NoSuchInterfaceException,
         IllegalBindingException, IllegalLifeCycleException {
      if (ITF_CLIENT_1.equals(clientItfName)) {
         i1 = (Itf1) serverItf;
      }
      // TODO: Bind the new client interface
      else {
         throw new NoSuchInterfaceException(clientItfName);
      }
   }

   public String[] listFc() {
      return new String[] { ITF_CLIENT_1
      //TODO: Add the new client interface name
      };
   }

   public Object lookupFc(String clientItfName) throws NoSuchInterfaceException {
      if (ITF_CLIENT_1.equals(clientItfName)) {
         return i1;
      }
      // TODO: Return the interface to which the new client interface is bound
      else {
         throw new NoSuchInterfaceException(clientItfName);
      }
   }

   public void unbindFc(String clientItfName) throws NoSuchInterfaceException, IllegalBindingException,
         IllegalLifeCycleException {
      if (ITF_CLIENT_1.equals(clientItfName)) {
         i1 = null;
      }
      // TODO: Unbind the new client interface
      else {
         throw new NoSuchInterfaceException(clientItfName);
      }
   }
}
```

Then, Master's ADL file needs also to be modified:

```
<!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.userguide.components.adl.interfaces.adl.Master">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Runner" role="server" na
>

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf1" role="client" name="
>
  <!-- TODO: Add the new client interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2 -->

  <content class="org.objectweb.proactive.examples.userguide.components.adl.interfaces.MasterImpl"/>

  <controller desc="primitive"/>
</definition>
```

Finally, you have to add a binding in the Composite's ADL file:

```
<!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.userguide.components.adl.interfaces.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Runner" role="server" na
>


 <component name="Master" definition="org.objectweb.proactive.examples.userguide.components.adl.interfaces.adl.Ma
>

 <component name="Slave" definition="org.objectweb.proactive.examples.userguide.components.adl.interfaces.adl.Slav
>

  <binding client="this.runner" server="Master.runner"/>
  <binding client="Master.i1" server="Slave.i1"/>
  <!-- TODO: Do the binding for the new interface -->

  <controller desc="composite"/>
</definition>
```

Thus, we propose you to:

1. Make the SlaveImpl class implement the Itf2 interface
2. Modify the Slave's ADL file to add the new interface
3. Modify method implementation of the BindingController methods in the MasterImpl class
4. Modify the Master's ADL file to add the new interface
5. Modify the Composite's ADL file to add a binding between the two new interfaces

3.2.4.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.adl.interfaces
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the adl-interfaces.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node438557290Slave:
 org.objectweb.proactive.examples.userguide.components.adl.interfaces.SlaveImpl@97aaa6
arg: hello - world

rmi://kisscool.inria.fr:6618/Node438557290Slave:
 org.objectweb.proactive.examples.userguide.components.adl.interfaces.SlaveImpl@97aaa6
I do nothing
```

3.2.4.4. Solutions and full code

Here is the solution:

```
package org.objectweb.proactive.examples.userguide.components.adl.interfaces;

import java.util.List;

import org.objectweb.proactive.api.PAActiveObject;


/**
 * @author The ProActive Team
 */
public class SlaveImpl implements Itf1
//TODO: Add the new interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2
      , Itf2
{

   public void compute(List<String> arg) {
      String str = "\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this + "\n";
      str += "arg: ";
      for (int i = 0; i < arg.size(); i++) {
         str += arg.get(i);
         if (i + 1 < arg.size()) {
            str += " - ";
         }
      }
      System.err.println(str);
   }

   public void doNothing() {
      System.err.println("\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this +
         "\nI do nothing");
   }
}
```

```
<!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.userguide.components.adl.interfaces.adl.Slave">
```

```
 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf1" role="server" name=
>
  <!-- TODO: Add the server interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2 -->

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2" role="server" name=
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.interfaces.SlaveImpl"/>

  <controller desc="primitive"/>
</definition>
```

```
package org.objectweb.proactive.examples.userguide.components.adl.interfaces;

import java.util.List;

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;


/**
 * @author The ProActive Team
 */
public class MasterImpl implements Runner, BindingController {
    public static String ITF_CLIENT_1 = "i1";
    public static String ITF_CLIENT_2 = "i2";
    private Itf1 i1;
    private Itf2 i2;

    public void run(List<String> arg) {
        i1.compute(arg);
        i2.doNothing();
    }

    public void bindFc(String clientItfName, Object serverItf) throws NoSuchInterfaceException,
            IllegalBindingException, IllegalLifeCycleException {
        if (ITF_CLIENT_1.equals(clientItfName)) {
            i1 = (Itf1) serverItf;
        }
        // TODO: Bind the new client interface
        else if (ITF_CLIENT_2.equals(clientItfName)) {
            i2 = (Itf2) serverItf;
        }
        else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

    public String[] listFc() {
        return new String[] { ITF_CLIENT_1
        //TODO: Add the new client interface name
```

```
            , ITF_CLIENT_2
        };
    }

    public Object lookupFc(String clientItfName) throws NoSuchInterfaceException {
        if (ITF_CLIENT_1.equals(clientItfName)) {
            return i1;
        }
        // TODO: Return the interface to which the new client interface is bound
        else if (ITF_CLIENT_2.equals(clientItfName)) {
            return i2;
        }
        else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

    public void unbindFc(String clientItfName) throws NoSuchInterfaceException, IllegalBindingException,
            IllegalLifeCycleException {
        if (ITF_CLIENT_1.equals(clientItfName)) {
            i1 = null;
        }
        // TODO: Unbind the new client interface
        else if (ITF_CLIENT_2.equals(clientItfName)) {
            i2 = null;
        }
        else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }
}
```

```
<!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.userguide.components.adl.interfaces.adl.Master">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Runner" role="server" na
>

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf1" role="client" name="
>
  <!-- TODO: Add the new client interface org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2 -->

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Itf2" role="client" name="
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.interfaces.MasterImpl"/>

  <controller desc="primitive"/>
</definition>
```

```
<!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.userguide.components.adl.interfaces.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.interfaces.Runner" role="server" na
>


 <component name="Master" definition="org.objectweb.proactive.examples.userguide.components.adl.interfaces.adl.Ma
>

 <component name="Slave" definition="org.objectweb.proactive.examples.userguide.components.adl.interfaces.adl.Slav
>

  <binding client="this.runner" server="Master.runner"/>
  <binding client="Master.i1" server="Slave.i1"/>
  <!-- TODO: Do the binding for the new interface -->
  <binding client="Master.i2" server="Slave.i2"/>

  <controller desc="composite"/>
</definition>
```

3.2.5. Multicast tutorial

3.2.5.1. Tutorial description

This tutorial will explain you how to create a multicast interface in order to get a component looking like this:

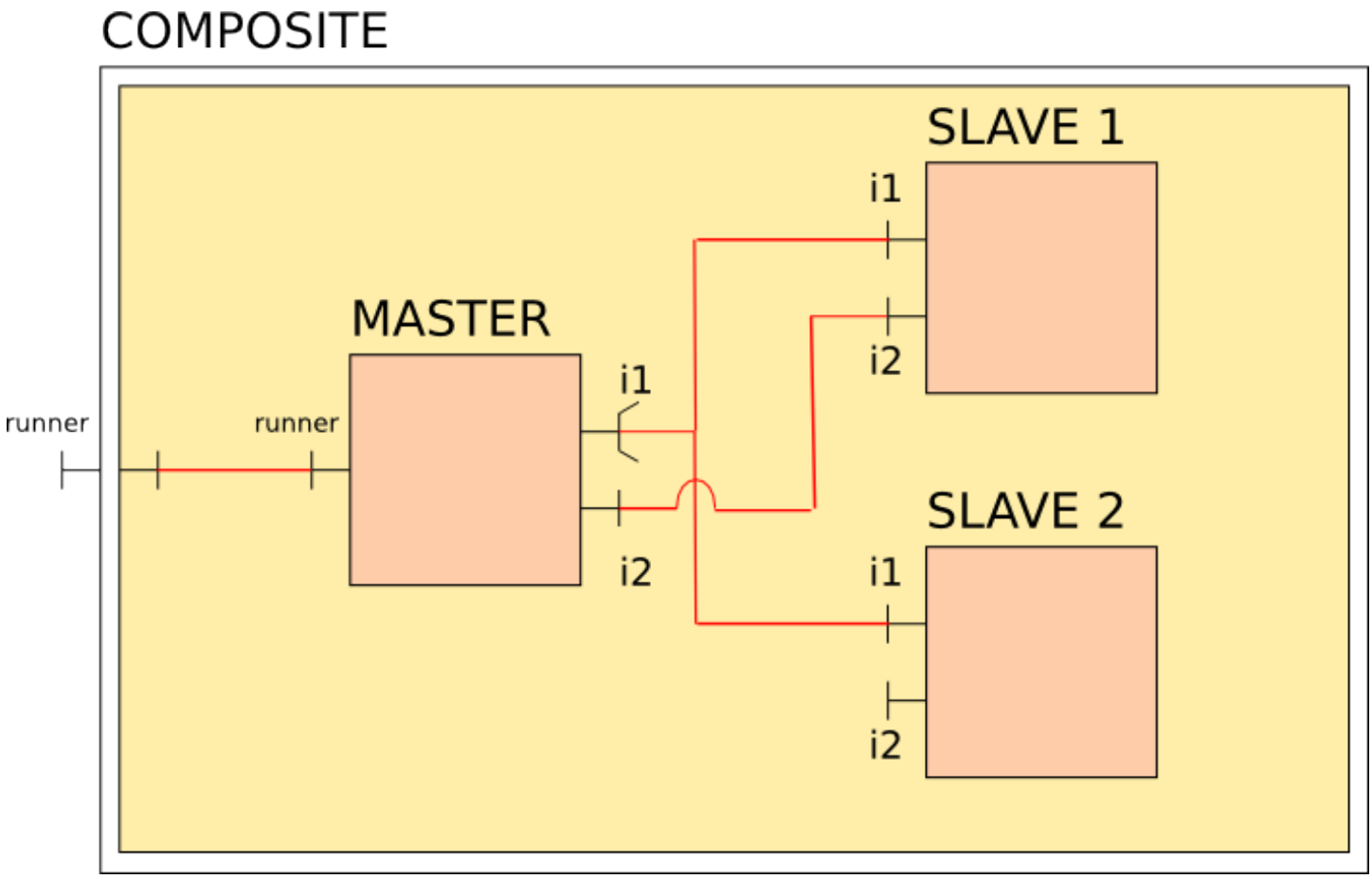


Figure 3.4. Composite component with a multicast client interface

For this, a new interface called Itf1Multicast will be added and will be used for the client interface of the Master component.

```
package org.objectweb.proactive.examples.userguide.components.adl.multicast;

import java.util.List;


/**
 * @author The ProActive Team
 */
public interface Itf1Multicast {
   // Optional TODO: Change the dispatch parameters policy to Round Robin
   void compute(List<String> arg);
}
```

Except this new interface and the replacement of

```
private Itf1 i1
```

by

```
private Itf1Multicast i1
```

in the MasterImpl class, there is no need of changes in Java code source. Only changes in ADL files are needed.

3.2.5.2. Proposed work

First, you have to change the client interface into the Master's ADL file in order to refer to the Itf1Multicast signature. You also have to change its cardinality to indicate that this interface is now a multicast interface.

```
<!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.userguide.components.adl.multicast.adl.Master">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Runner" role="server" nar
>

  <!-- TODO: Add the client multicast interface
 org.objectweb.proactive.examples.userguide.components.adl.multicast.Itf1Multicast -->
  <!-- Note: do not forget to set its cardinality -->


 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Itf2" role="client" name="i
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.multicast.MasterImpl"/>

  <controller desc="primitive"/>
</definition>
```

Then, you have to modify the Composite's ADL file to add a new Slave component and to add also some bindings.

```
<!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.userguide.components.adl.multicast.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Runner" role="server" nar
>


 <component name="Master" definition="org.objectweb.proactive.examples.userguide.components.adl.multicast.adl.Mas
>

 <component name="Slave1" definition="org.objectweb.proactive.examples.userguide.components.adl.multicast.adl.Slav
>

  <!-- TODO: Add a second Slave Component -->

  <binding client="this.runner" server="Master.runner"/>

  <!-- TODO: Do the binding between the Master Component and the Slave Components on the Multicast Interface -->

  <binding client="Master.i2" server="Slave1.i2"/>

  <controller desc="composite"/>
</definition>
```

Thus, we propose you to:

1. Modify the client interface of the Master component
2. Add a Slave component into the composite
3. Add bindings to realise the multicast
4. Optional: Change the dispatch parameter policy in the multicast interface to round robin dispatch mode

3.2.5.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.adl.multicast
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the adl-interfaces.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node1434753324Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@3861e6
arg: hello - world

rmi://kisscool.inria.fr:6618/Node1434753324Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@1132e76
arg: hello - world

rmi://kisscool.inria.fr:6618/Node1434753324Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@3861e6
I do nothing
```

If you have done the optional work, you should have a display as follows:

```
rmi://kisscool.inria.fr:6618/Node1653076599Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@789d63
arg: hello

rmi://kisscool.inria.fr:6618/Node1653076599Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@a4ed99
arg: world

rmi://kisscool.inria.fr:6618/Node1653076599Slave:
 org.objectweb.proactive.examples.userguide.components.adl.multicast.SlaveImpl@789d63
I do nothing
```

3.2.5.4. Solutions and full code

Here are solutions for the ADL files:

```
<!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.userguide.components.adl.multicast.adl.Master">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Runner" role="server" nam
>
```

```
 <!-- TODO: Add the client multicast interface
org.objectweb.proactive.examples.userguide.components.adl.multicast.Itf1Multicast -->
 <!-- Note: do not forget to set its cardinality -->

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Itf1Multicast" role="client"
>


 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Itf2" role="client" name="i
>

 <content class="org.objectweb.proactive.examples.userguide.components.adl.multicast.MasterImpl"/>

 <controller desc="primitive"/>
</definition>
```

```
<!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.userguide.components.adl.multicast.adl.Composite">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.multicast.Runner" role="server" nar
>


 <component name="Master" definition="org.objectweb.proactive.examples.userguide.components.adl.multicast.adl.Mas
>

 <component name="Slave1" definition="org.objectweb.proactive.examples.userguide.components.adl.multicast.adl.Slav
>

 <!-- TODO: Add a second Slave Component -->

 <component name="Slave2" definition="org.objectweb.proactive.examples.userguide.components.adl.multicast.adl.Slav
>

 <binding client="this.runner" server="Master.runner"/>

 <!-- TODO: Do the binding between the Master Component and the Slave Components on the Multicast Interface -->
 <binding client="Master.i1" server="Slave1.i1"/>
 <binding client="Master.i1" server="Slave2.i1"/>

 <binding client="Master.i2" server="Slave1.i2"/>

 <controller desc="composite"/>
</definition>
```

If you do not want to do the optional work, you do not have to modify other files. Otherwise, if you want to change the dispatch parameter policy, you have to make some little modification in Java code sources. The first one is to change the signature of the compute method in the Itf1Multicast interface. Change the line

```
void compute(List<String> arg);
```

by

```
void compute(@ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN) List<String> arg);
```

Then, you have to change the Itf1 interface since the server interface will no longer receive a list of String but instead, just a String. Thus, in the Itf1 interface, change the line

```
void compute(List<String> arg);
```

by

```
void compute(String arg);
```

and in the SlaveImpl class, change the implementation of compute so that it returns a String:

```
public void compute(String arg) {
  String str = "\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this + "\n";
  str += "arg: " + arg;
  System.err.println(str);
}
```

3.2.6. Deployment tutorial

3.2.6.1. Tutorial description

Now that we know how to use component, we would like to benefit from active object properties. One of these is the deployment on several Java Virtual Machines (JVM). This tutorial will explain you how to deploy our component on 3 differents JVM. For this, we will use GCM deployment model. If you have done the active object tutorials, you should know what this model is and how to use it. If not, please refer to Chapter 21. ProActive Grid Component Model Deployment[2].

For the deployment, we will use the following application description:

```
<GCMApplication xmlns="urn:gcm:application:1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www-sop.inria.fr/oasis/ProActive/schemas http://proactive.inria.fr/schemas/
gcm/1.0/ApplicationDescriptorSchema.xsd">

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

   <application>
      <proactive base="root" relpath="${proactive.home}">
         <configuration>
            <java base="root" relpath="${java.home}/bin/java" />
            <applicationClasspath>
               <pathElement base="root" relpath="${proactive.home}/classes" />
            </applicationClasspath>
         </configuration>
         <virtualNode id="master-node" capacity="1">
            <nodeProvider refid="MASTER_REMOTE_PROVIDER" />
```

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/GCMDeployment.html#GCMDeployment

```
        </virtualNode>
        <virtualNode id="slave-node" capacity="2">
          <nodeProvider refid="SLAVE_REMOTE_PROVIDER" />
        </virtualNode>
      </proactive>
    </application>

    <resources>
      <nodeProvider id="MASTER_REMOTE_PROVIDER">
        <file path="GCMD_Local.xml" />
      </nodeProvider>
      <nodeProvider id="SLAVE_REMOTE_PROVIDER">
        <file path="GCMD_Local.xml" />
      </nodeProvider>
    </resources>
</GCMApplication>
```

which refers to the following deployment descriptor, used both for the deployment of the Master component and for the Slave components:

```
<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" />
    <javaPropertyVariable name="user.name" />
    <javaPropertyDescriptorDefault name="os" value="windows" />
  </environment>

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

  <infrastructure>

    <hosts>
      <host id="hLocalhost" os="${os}" hostCapacity="3" vmCapacity="1">
        <homeDirectory base="root" relpath="${user.home}" />
      </host>

    </hosts>

  </infrastructure>
</GCMDeployment>
```

The main things that change comparing to the previous tutorial are the Main method where deployment has to be loaded and used, and ADL files into which virtual node of the corresponding component deployment has to be added.

3.2.6.2. Proposed work

First, you have to add a virtual-node tag into the Slave's ADL file to indicate where to deploy the Slave components:

```
<!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.userguide.components.adl.deployment.adl.Slave">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.deployment.Itf1" role="server" name
>

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.deployment.Itf2" role="server" name
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.deployment.SlaveImpl"/>

  <controller desc="primitive"/>

  <!-- TODO: Affect this Component to the virtual node "slave-node" -->
</definition>
```

Then, you have to modify the Main method so as to load and use the application descriptor.

```
package org.objectweb.proactive.examples.userguide.components.adl.deployment;

import java.io.File;
import java.net.URL;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.proactive.core.component.adl.FactoryFactory;
import org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment;
import org.objectweb.proactive.gcmdeployment.GCMApplication;
import org.objectweb.proactive.gcmdeployment.GCMVirtualNode;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {

      // TODO: Load the Application Descriptor

      // TODO: Start the deployment

      Factory factory = FactoryFactory.getFactory();

      Map<String, Object> context = new HashMap<String, Object>();

      // TODO: Put the Application Descriptor in the context

      Component composite = (Component) factory
```

```
            .newComponent(
                "org.objectweb.proactive.examples.userguide.components.adl.deployment.adl.Composite",
                context);

        GCM.getGCMLifeCycleController(composite).startFc();

        Runner runner = (Runner) composite.getFcInterface("runner");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        runner.run(arg);

        GCM.getGCMLifeCycleController(composite).stopFc();

        System.exit(0);
    }
}
```

Thus, we propose you to:

1. Add a virtual-node tag in the Slave's ADL file. Hint: draw your inspiration on the Master's ADL file.
2. Complete the Main method. Hint: Look at the deployment of active objects

3.2.6.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.adl.deployment
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the adl-deployment.[sh|bat] script. You should then see a display looking like this:

```
--- User Guide: ADL Deployment -----------------------------------------------
30447@kisscool - [INFO oactive.remoteobject] Remote Object Factory provider <pamr, class
 org.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory> found
30447@kisscool - [INFO    communication.rmi] Created a new registry on port 6618
30447@kisscool - [INFO        proactive.mop] Generating class :
 pa.stub.org.objectweb.proactive.gcmdeployment._StubGCMVirtualNode
30447@kisscool - [INFO        proactive.mop] Generating class : pa.stub.org.objectweb.lector -
Djava.security.policy="/home/ffonteno/proactive-git/programming/tutorials/scripts/proactive.java.policy"
 org.objectweb.proactive.core.runtime.StartPARuntime -p rmi://kisscool.inria.fr:6618/PA_JVM798723297 -c 1 -i 1 -d
 8194810407958514089 -b http://kisscool.inria.fr:49352/classServer/  '
30447@kisscool - [INFO                job.1] executing command=/usr/bin/ssh -l ffonteno localhost "'/home/
ffonteno/src/java/jdk1.5.0_17/jre/bin/java" -cp "/home/ffonteno/proactive-git/programming/tutorials/dist/lib/
ProActive.jar:/home/ffonteno/proactive-git/programming/tutorials/classes" -Dproactive.log4j.collector=rmi://
kisscool.inria.fr:6618/8194810407958514089/logCollector -Djava.security.policy="/home/ffonteno/proactive-git/
programming/tutorials/scripts/proactive.java.policy" org.objectweb.proactive.core.runtime.StartPARuntime -p rmi://
kisscool.inria.fr:6618/PA_JVM798723297 -c 1 -i 1 -d 8194810407958514089 -b http://kisscool.inria.fr:49352/classServer/
 '
30447@kisscool - [INFO      deployment.GCMD] Starting group id=gCluster #commands=1
30447@kisscool - [INFO                job.2] executing command=/usr/bin/ssh -l ffonteno localhost "'/home/
ffonteno/src/java/jdk1.5.0_17/jre/bin/java" -cp "/home/ffonteno/proactive-git/programming/tutorials/dist/lib/
ProActive.jar:/home/ffonteno/proactive-git/programming/tutorials/classes" -Dproactive.log4j.collector=rmi://
kisscool.inria.fr:6618/8194810407958514089/logCollector -Djava.security.policy="/home/ffonteno/proactive-git/
programming/tutorials/scripts/proactive.java.policy" org.objectweb.proactive.core.runtime.StartPARuntime -p rmi://
```

```
kisscool.inria.fr:6618/PA_JVM798723297 -c 1 -i 2 -d 8194810407958514089 -b http://kisscool.inria.fr:49352/classServer/
 '
30447@kisscool - [INFO       proactive.mop] Generating class :
 pa.stub.org.objectweb.proactive.core.util.log.remote._StubProActiveLogCollector
30447@kisscool - [INFO       proactive.mop] Generating class :
 pa.stub.org.objectweb.proactive.examples.userguide.components.adl.deployment._StubMasterImpl
30447@kisscool - [INFO       proactive.mop] Generating class :
 pa.stub.org.objectweb.proactive.examples.userguide.components.adl.deployment._StubSlaveImpl
30447@kisscool - [INFO                job.1]
30447@kisscool - [INFO                job.1] rmi://kisscool.inria.fr:6618/PA_JVM539286369_GCMNode-0Slave:
 org.objectweb.proactive.examples.userguide.components.adl.deployment.SlaveImpl@e931d9
30447@kisscool - [INFO                job.1] arg: hello - world
30447@kisscool - [INFO                job.0]
30447@kisscool - [INFO                job.0] rmi://kisscool.inria.fr:6618/PA_JVM1043221418_GCMNode-0Slave:
 org.objectweb.proactive.examples.userguide.components.adl.deployment.SlaveImpl@e9a7c2
30447@kisscool - [INFO                job.0] arg: hello - world
30447@kisscool - [INFO                job.0]
30447@kisscool - [INFO                job.0] rmi://kisscool.inria.fr:6618/PA_JVM1043221418_GCMNode-0Slave:
 org.objectweb.proactive.examples.userguide.components.adl.deployment.SlaveImpl@e9a7c2
30447@kisscool - [INFO                job.0] I do nothing
```

3.2.6.4. Solutions and full code

Here are the solutions:

```
<!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.userguide.components.adl.deployment.adl.Slave">

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.deployment.Itf1" role="server" name
>

 <interface signature="org.objectweb.proactive.examples.userguide.components.adl.deployment.Itf2" role="server" name
>

  <content class="org.objectweb.proactive.examples.userguide.components.adl.deployment.SlaveImpl"/>

  <controller desc="primitive"/>

  <!-- TODO: Affect this Component to the virtual node "slave-node" -->
  <virtual-node name="slave-node"/>
</definition>
```

```
package org.objectweb.proactive.examples.userguide.components.adl.deployment;

import java.io.File;
import java.net.URL;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
```

```
import org.objectweb.fractal.api.Component;
import org.objectweb.proactive.core.component.adl.FactoryFactory;
import org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment;
import org.objectweb.proactive.gcmdeployment.GCMApplication;
import org.objectweb.proactive.gcmdeployment.GCMVirtualNode;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {

      // TODO: Load the Application Descriptor
      String descriptorPath = "file://" +
         System.getProperty("proactive.home") +
         "/src/Examples/org/objectweb/proactive/examples/userguide/components/adl/deployment/descriptors/
application_descriptor.xml";
      // This tricky line enables to use this path in both Linux and Windows OS
      File deploymentFile = new File((new URL(descriptorPath)).toURI().getPath());
      GCMApplication gcma = PAGCMDeployment.loadApplicationDescriptor(deploymentFile);

      // TODO: Start the deployment
      gcma.startDeployment();
      gcma.waitReady();
      GCMVirtualNode vn = gcma.getVirtualNode("slave-node");
      vn.waitReady();

      Factory factory = FactoryFactory.getFactory();

      Map<String, Object> context = new HashMap<String, Object>();

      // TODO: Put the Application Descriptor in the context
      context.put("deployment-descriptor", gcma);

      Component composite = (Component) factory
            .newComponent(
                  "org.objectweb.proactive.examples.userguide.components.adl.deployment.adl.Composite",
                  context);

      GCM.getGCMLifeCycleController(composite).startFc();

      Runner runner = (Runner) composite.getFcInterface("runner");
      List<String> arg = new ArrayList<String>();
      arg.add("hello");
      arg.add("world");
      runner.run(arg);

      GCM.getGCMLifeCycleController(composite).stopFc();

      System.exit(0);
   }
}
```

3.3. Create and use components using the API

3.3.1. Introduction

This second part of the tutorial shows you how to use the API in order to construct GCM components. This part is composed of three exercises:

- **Starter** - this exercise make you create a simple primitive component called Slave
- **Composite** - this exercise make you create a composite component composed of a Master and a Slave component.
- **Interfaces** - this exercise make you modify the previous Slave component to have it implement two server interfaces.

3.3.2. Starter tutorial

3.3.2.1. Tutorial description

This tutorial explains how to create a simple component named Slave using the API. This component looks like that:

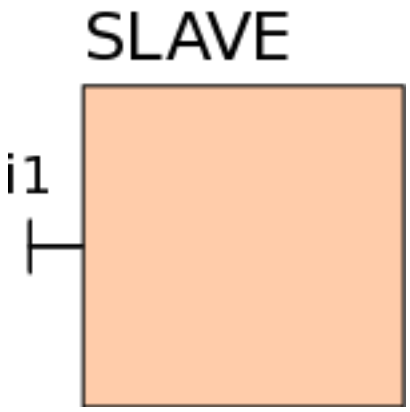


Figure 3.5. Slave component

The interface used for this example is the following one:

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

import java.util.List;


/**
 * @author The ProActive Team
 */
public interface Itf1 {
    void compute(List<String> arg);
}
```

This interface defines a compute() method which is implemented in the following SlaveImpl class:

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

import java.util.List;
```

```
import org.objectweb.proactive.api.PAActiveObject;


/**
 * @author The ProActive Team
 */
public class SlaveImpl implements Itf1 {

    public void compute(List<String> arg) {
        String str = "\n" + PAActiveObject.getBodyOnThis().getNodeURL() + "Slave: " + this + "\n";
        str += "arg: ";
        for (int i = 0; i < arg.size(); i++) {
            str += arg.get(i);
            if (i + 1 < arg.size()) {
                str += " - ";
            }
        }
        System.err.println(str);
    }

}
```

These two classes are strictly the same as the ones used in the Starter tutorial using ADL files. In this tutorial, instead of describing the component into an ADL file, we will describe it directly in the main() method.

3.3.2.2. Proposed work

Here is the main() method that you have to complete:

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

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

    public static void main(String[] args) throws Exception {

        // TODO: Get the Bootstrap Component
```

```
        // TODO: Get the TypeFactory
        // TODO: Get the GenericFactory
        // TODO: Create the i1 Interface Type (org.objectweb.proactive.examples.userguide.components.api.starter.Itf1)
        // TODO: Create the Slave Component type
        // TODO: Create the Slave Component

        GCM.getGCMLifeCycleController(slave).startFc();

        Itf1 itf1 = (Itf1) slave.getFcInterface("i1");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        itf1.compute(arg);

        GCM.getGCMLifeCycleController(slave).stopFc();

        System.exit(0);
    }
}
```

Thus, we propose you to:

1. Get the Bootstrap component using the org.objectweb.proactive.core.component.Utils.getBootstrapComponent() method
2. Get the GCMTypeFactory using the GCM.getGCMTypeFactory() method
3. Get the GenericFactory using the GCM.getGenericFactory() method
4. Create the i1 Interface Type (org.objectweb.proactive.examples.userguide.components.api.starter.Itf1) using the createFcItfType() method of the GCMTypeFactory class
5. Create the Slave Component type using the createFcType() method of the GCMTypeFactory class
6. Create the Slave Component instance using the newFcInstance() method of the GenericFactory class

3.3.2.3. Test your work

To build your work, go to the compile directory into the tutorials directory (not the one located into the ProActive directory) and type:

```
build components.api.starter
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the api-starter.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node564429316Slave:
 org.objectweb.proactive.examples.userguide.components.api.starter.SlaveImpl@109da93
arg: hello - world
```

3.3.2.4. Solutions and full code

Here is how you should have filled in the main() method:

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

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
```

```
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {

      // TODO: Get the Bootstrap Component
      Component boot = Utils.getBootstrapComponent();
      // TODO: Get the TypeFactory
      GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
      // TODO: Get the GenericFactory
      GenericFactory gf = GCM.getGenericFactory(boot);
      // TODO: Create the i1 Interface Type (org.objectweb.proactive.examples.userguide.components.api.starter.Itf1)
      InterfaceType itf1Slave = tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.SERVER,
            TypeFactory.MANDATORY, TypeFactory.SINGLE);
      // TODO: Create the Slave Component type
      ComponentType tSlave = tf.createFcType(new InterfaceType[] { itf1Slave });
      // TODO: Create the Slave Component
      Component slave = gf.newFcInstance(tSlave, new ControllerDescription("slave", Constants.PRIMITIVE),
            SlaveImpl.class.getName());

      GCM.getGCMLifeCycleController(slave).startFc();

      Itf1 itf1 = (Itf1) slave.getFcInterface("i1");
      List<String> arg = new ArrayList<String>();
      arg.add("hello");
      arg.add("world");
      itf1.compute(arg);

      GCM.getGCMLifeCycleController(slave).stopFc();

      System.exit(0);
   }
}
```

3.3.3. Composite tutorial

3.3.3.1. Tutorial description

This tutorial shows how to create a composite component composed of a Master and a Slave component.

This composite can be represented as follows:

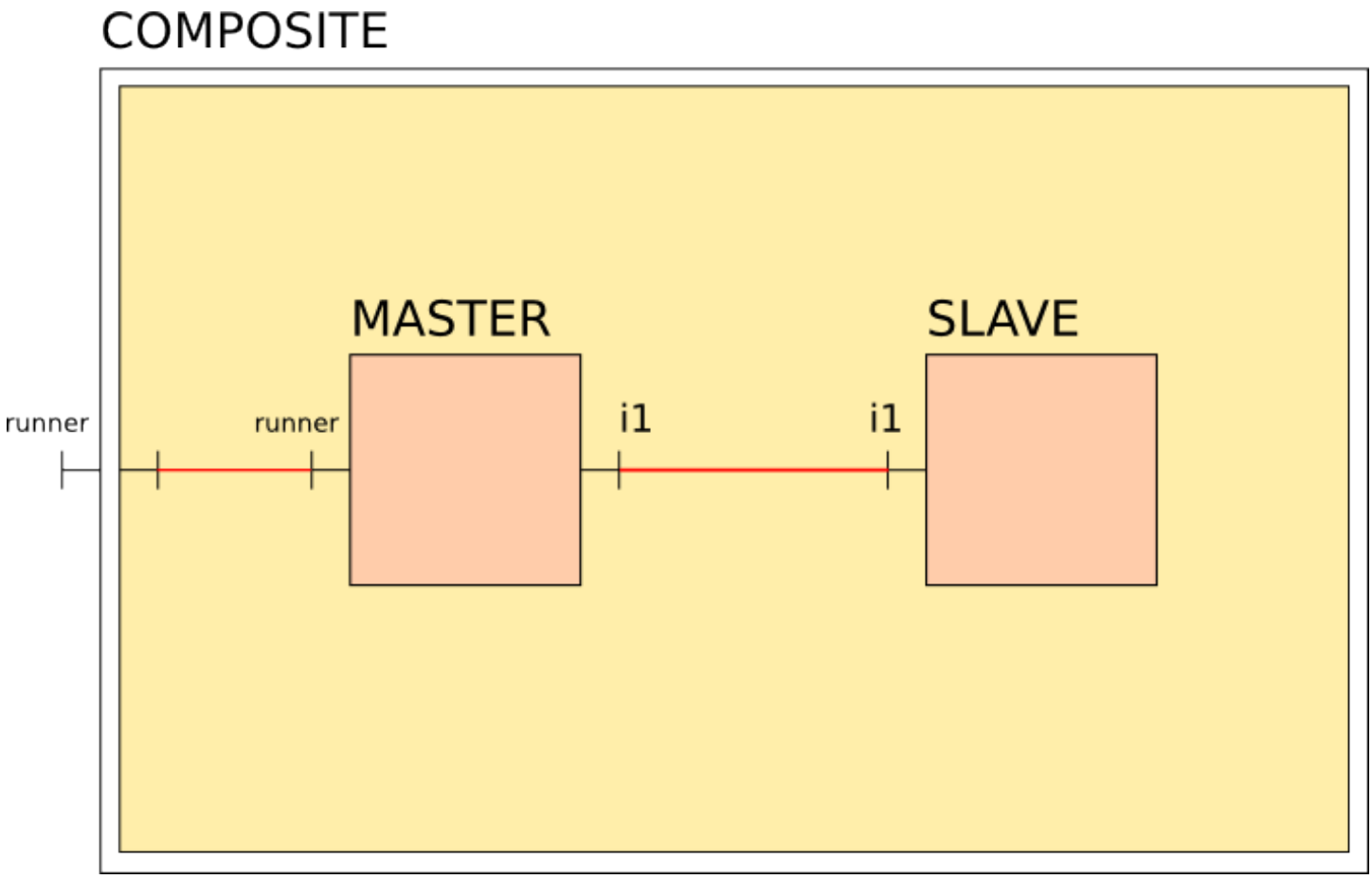


Figure 3.6. Composite component

The Slave component is exactly the same as in the previous tutorial. It is defined by the Itf1 interface implemented by the SlaveImpl class.

Concerning the Master component, it is defined by the following Runner interface:

```
package org.objectweb.proactive.examples.userguide.components.api.composite;

import java.util.List;


/**
 * @author The ProActive Team
 */
public interface Runner {
    public void run(List<String> arg);
}
```

which is implemented by the MasterImpl class:

```
package org.objectweb.proactive.examples.userguide.components.api.composite;

import java.util.List;
```

```
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;


/**
 * @author The ProActive Team
 */
public class MasterImpl implements Runner, BindingController {
    public static String ITF_CLIENT = "i1";
    private Itf1 i1;

    public void run(List<String> arg) {
        i1.compute(arg);
    }

    public void bindFc(String clientItfName, Object serverItf) throws NoSuchInterfaceException,
            IllegalBindingException, IllegalLifeCycleException {
        if (ITF_CLIENT.equals(clientItfName)) {
            i1 = (Itf1) serverItf;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

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

    public Object lookupFc(String clientItfName) throws NoSuchInterfaceException {
        if (ITF_CLIENT.equals(clientItfName)) {
            return i1;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }

    public void unbindFc(String clientItfName) throws NoSuchInterfaceException, IllegalBindingException,
            IllegalLifeCycleException {
        if (ITF_CLIENT.equals(clientItfName)) {
            i1 = null;
        } else {
            throw new NoSuchInterfaceException(clientItfName);
        }
    }
}
```

This class contains an Itf1 member representing the client interface which is used in the BindingController method. These methods are:

- bindFc() - used to bind interfaces
- listFc() - used to list interfaces
- lookupFc() - used to get one of the available interfaces using its name
- unbindFc() - used to unbind interfaces

The only things that remain to do now so as to be able to use our composite component are first, to define all the three components (Slave, Master and Composite) in our main() method using the Java API and then, to add the two sub-components (Master and Slave) to the Composite component and to write the two bindings.

3.3.3.2. Proposed work

Here is the Main class that you have to fill in:

```
package org.objectweb.proactive.examples.userguide.components.api.composite;

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.api.control.ContentController;
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {
      Component boot = Utils.getBootstrapComponent();
      GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
      GenericFactory gf = GCM.getGenericFactory(boot);
      ComponentType tComposite = tf.createFcType(new InterfaceType[] { tf.createFcItfType("runner",
            Runner.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });

      // TODO: Create the Master Component type

      ComponentType tSlave = tf.createFcType(new InterfaceType[] { tf.createFcItfType("i1", Itf1.class
            .getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });

      Component slave = gf.newFcInstance(tSlave, new ControllerDescription("slave", Constants.PRIMITIVE),
            SlaveImpl.class.getName());

      // TODO: Create the Master Component

      Component composite = gf.newFcInstance(tComposite, new ControllerDescription("composite",
         Constants.COMPOSITE), null);

      // TODO: Add slave and master components to the composite component
```

```
        // TODO: Do the bindings

        GCM.getGCMLifeCycleController(composite).startFc();

        Runner runner = (Runner) composite.getFcInterface("runner");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        runner.run(arg);

        GCM.getGCMLifeCycleController(composite).stopFc();

        System.exit(0);
    }
}
```

We propose you to:

1. Create the Master component type as you created the Slave component type in the previous tutorial. Hint: in that case, there are two interfaces (one client and one server interface)
2. Instantiate the Master component
3. Add the two sub-components (Master and Slave) to the Composite component. Hint: use the GCM.getContentController() static method to get a ContentController and use it to add these sub-components.
4. Do the bindings. Hint: For doing a binding, you should first get the BindingController of the component containing the client interface you want to bind and then, call the bindFc() method on it. To get the BindingController, use the GCM.getBindingController() method.

3.3.3.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.api.composite
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the api-composite.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node480011239Slave:
 org.objectweb.proactive.examples.userguide.components.api.composite.SlaveImpl@c62080
arg: hello - world
```

3.3.3.4. Solutions and full code

Here is how you should have filled in the Main class:

```
package org.objectweb.proactive.examples.userguide.components.api.composite;

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.api.control.ContentController;
```

```
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {
      Component boot = Utils.getBootstrapComponent();
      GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
      GenericFactory gf = GCM.getGenericFactory(boot);
      ComponentType tComposite = tf.createFcType(new InterfaceType[] { tf.createFcItfType("runner",
            Runner.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });

      // TODO: Create the Master Component type
      ComponentType tMaster = tf.createFcType(new InterfaceType[] {
            tf.createFcItfType("runner", Runner.class.getName(), TypeFactory.SERVER,
                  TypeFactory.MANDATORY, TypeFactory.SINGLE),
            tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.CLIENT, TypeFactory.MANDATORY,
                  TypeFactory.SINGLE) });

      ComponentType tSlave = tf.createFcType(new InterfaceType[] { tf.createFcItfType("i1", Itf1.class
            .getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });

      Component slave = gf.newFcInstance(tSlave, new ControllerDescription("slave", Constants.PRIMITIVE),
            SlaveImpl.class.getName());

      // TODO: Create the Master Component
      Component master = gf.newFcInstance(tMaster,
            new ControllerDescription("master", Constants.PRIMITIVE), MasterImpl.class.getName());

      Component composite = gf.newFcInstance(tComposite, new ControllerDescription("composite",
         Constants.COMPOSITE), null);

      // TODO: Add slave and master components to the composite component
      ContentController cc = GCM.getContentController(composite);
      cc.addFcSubComponent(slave);
      cc.addFcSubComponent(master);

      // TODO: Do the bindings
      BindingController bcMaster = GCM.getBindingController(master);
      bcMaster.bindFc("i1", slave.getFcInterface("i1"));
      BindingController bcComposite = GCM.getBindingController(composite);
      bcComposite.bindFc("runner", master.getFcInterface("runner"));

      GCM.getGCMLifeCycleController(composite).startFc();
```

```
        Runner runner = (Runner) composite.getFcInterface("runner");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        runner.run(arg);

        GCM.getGCMLifeCycleController(composite).stopFc();

        System.exit(0);
    }
}
```

3.3.4. Interface tutorial

3.3.4.1. Tutorial description

In this tutorial, we will add a new server interface to the Slave component and bind it to a new client interface of the Master component. Our composite component will therefore look like as follows:

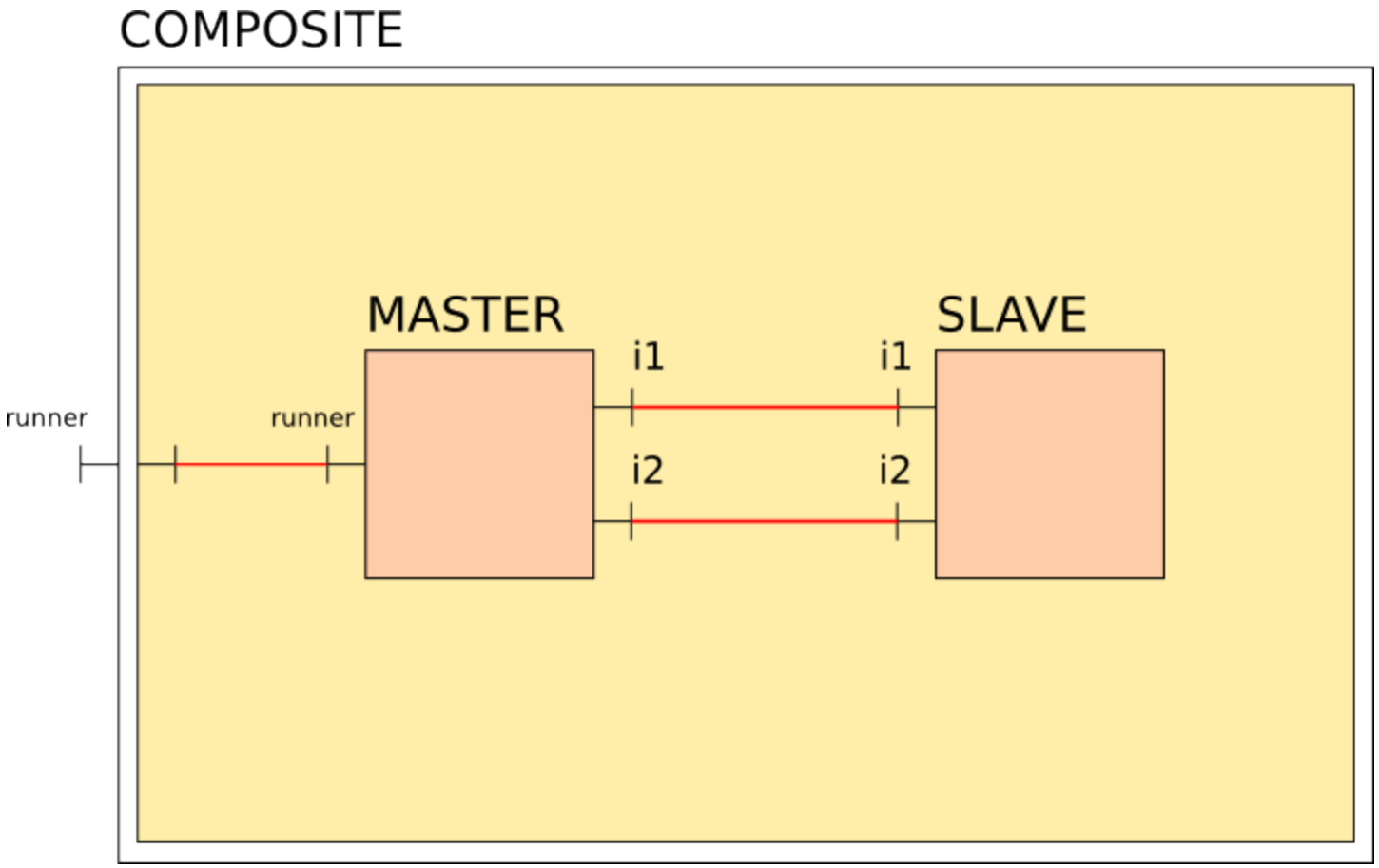


Figure 3.7. Composite component with the new interface

We introduce a second interface named Itf2 which source code is the following one:

```
package org.objectweb.proactive.examples.userguide.components.api.interfaces;
```

```
/**
 * @author The ProActive Team
 */
public interface Itf2 {
    void doNothing();
}
```

The aim of this tutorial is to make you modify the previous code in order to add this new interface using the API. All the necessary modification are made into the Main class.

3.3.4.2. Proposed work

Here is the main() method that you have to complete:

```
package org.objectweb.proactive.examples.userguide.components.api.interfaces;

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.api.control.ContentController;
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

    public static void main(String[] args) throws Exception {
        Component boot = Utils.getBootstrapComponent();
        GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
        GenericFactory gf = GCM.getGenericFactory(boot);
        ComponentType tComposite = tf.createFcType(new InterfaceType[] { tf.createFcItfType("runner",
                Runner.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });
        ComponentType tMaster = tf.createFcType(new InterfaceType[] {
                tf.createFcItfType("runner", Runner.class.getName(), TypeFactory.SERVER,
                        TypeFactory.MANDATORY, TypeFactory.SINGLE),
                tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.CLIENT, TypeFactory.MANDATORY,
                        TypeFactory.SINGLE)
                //TODO: Add the new client interface
                });
        ComponentType tSlave = tf.createFcType(new InterfaceType[] {
                tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY,
                        TypeFactory.SINGLE)
                //TODO: Add the new server interface
```

```
            });
        Component slave = gf.newFcInstance(tSlave, new ControllerDescription("slave", Constants.PRIMITIVE),
                SlaveImpl.class.getName());
        Component master = gf.newFcInstance(tMaster,
                new ControllerDescription("master", Constants.PRIMITIVE), MasterImpl.class.getName());
        Component composite = gf.newFcInstance(tComposite, new ControllerDescription("composite",
           Constants.COMPOSITE), null);

        ContentController cc = GCM.getContentController(composite);
        cc.addFcSubComponent(slave);
        cc.addFcSubComponent(master);

        BindingController bcComposite = GCM.getBindingController(composite);
        bcComposite.bindFc("runner", master.getFcInterface("runner"));
        BindingController bcMaster = GCM.getBindingController(master);
        bcMaster.bindFc("i1", slave.getFcInterface("i1"));

        // TODO: Do the binding for the new interface

        GCM.getGCMLifeCycleController(composite).startFc();

        Runner runner = (Runner) composite.getFcInterface("runner");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        runner.run(arg);

        GCM.getGCMLifeCycleController(composite).stopFc();

        System.exit(0);
    }
}
```

Thus, we propose you to:

1. Add the new client interface in the Master component type
2. Add the new server interface in the Slave component type
3. Add a new binding between these two interfaces

3.3.4.3. Test your work

To build your work, go to the compile directory into the tutorials directory and type:

```
build components.api.interfaces
```

Once you get a successful compilation, go to the scripts/Components directory located into the tutorials one and launch the api-interfaces.[sh|bat] script. You should then see a display looking like this:

```
rmi://kisscool.inria.fr:6618/Node275638725Slave:
 org.objectweb.proactive.examples.userguide.components.api.interfaces.SlaveImpl@3aa791
arg: hello - world

rmi://kisscool.inria.fr:6618/Node275638725Slave:
 org.objectweb.proactive.examples.userguide.components.api.interfaces.SlaveImpl@3aa791
I do nothing
```

3.3.4.4. Solutions and full code

Here is the solution:

```
package org.objectweb.proactive.examples.userguide.components.api.interfaces;

import java.util.ArrayList;
import java.util.List;

import org.etsi.uri.gcm.api.type.GCMTypeFactory;
import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.fractal.api.control.ContentController;
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.proactive.core.component.Constants;
import org.objectweb.proactive.core.component.ControllerDescription;
import org.objectweb.proactive.core.component.Utils;


/**
 * @author The ProActive Team
 */
public class Main {

   public static void main(String[] args) throws Exception {
      Component boot = Utils.getBootstrapComponent();
      GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
      GenericFactory gf = GCM.getGenericFactory(boot);
      ComponentType tComposite = tf.createFcType(new InterfaceType[] { tf.createFcItfType("runner",
            Runner.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY, TypeFactory.SINGLE) });
      ComponentType tMaster = tf.createFcType(new InterfaceType[] {
            tf.createFcItfType("runner", Runner.class.getName(), TypeFactory.SERVER,
                  TypeFactory.MANDATORY, TypeFactory.SINGLE),
            tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.CLIENT, TypeFactory.MANDATORY,
                  TypeFactory.SINGLE)
            //TODO: Add the new client interface
            ,
            tf.createFcItfType("i2", Itf2.class.getName(), TypeFactory.CLIENT, TypeFactory.MANDATORY,
                  TypeFactory.SINGLE)
            });
      ComponentType tSlave = tf.createFcType(new InterfaceType[] {
            tf.createFcItfType("i1", Itf1.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY,
                  TypeFactory.SINGLE)
            //TODO: Add the new server interface
            ,
            tf.createFcItfType("i2", Itf2.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY,
                  TypeFactory.SINGLE)
            });
      Component slave = gf.newFcInstance(tSlave, new ControllerDescription("slave", Constants.PRIMITIVE),
            SlaveImpl.class.getName());
```

```
        Component master = gf.newFcInstance(tMaster,
                new ControllerDescription("master", Constants.PRIMITIVE), MasterImpl.class.getName());
        Component composite = gf.newFcInstance(tComposite, new ControllerDescription("composite",
            Constants.COMPOSITE), null);

        ContentController cc = GCM.getContentController(composite);
        cc.addFcSubComponent(slave);
        cc.addFcSubComponent(master);

        BindingController bcComposite = GCM.getBindingController(composite);
        bcComposite.bindFc("runner", master.getFcInterface("runner"));
        BindingController bcMaster = GCM.getBindingController(master);
        bcMaster.bindFc("i1", slave.getFcInterface("i1"));

        // TODO: Do the binding for the new interface
        bcMaster.bindFc("i2", slave.getFcInterface("i2"));

        GCM.getGCMLifeCycleController(composite).startFc();

        Runner runner = (Runner) composite.getFcInterface("runner");
        List<String> arg = new ArrayList<String>();
        arg.add("hello");
        arg.add("world");
        runner.run(arg);

        GCM.getGCMLifeCycleController(composite).stopFc();

        System.exit(0);
    }
}
```

Part II. Programming With Components

Table of Contents

Chapter 4. User guide

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

4.1. 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 [1]. 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).

4.1.1. 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 available in the Examples/org/objectweb/proactive/examples/components/helloworld directory:

```
 1:<?xml version="1.0" encoding="ISO-8859-1" ?>
 2:<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath://org/objectweb/
proactive/core/component/adl/xml/proactive.dtd">
 3:
 4:<definition name="org.objectweb.proactive.examples.components.helloworld.HelloWorld">
 5:
 <interface name="m" role="server" signature="org.objectweb.proactive.examples.components.helloworld.Main"/
>
 6:
 7:
 <component name="client" definition="org.objectweb.proactive.examples.components.helloworld.ClientImpl"/>
 8:    <component name="server">
 9:
 <interface name="s" role="server" signature="org.objectweb.proactive.examples.components.helloworld.Service"/
>
 10:       <content class="ServerImpl"/>
 11:       <attributes signature="org.objectweb.proactive.examples.components.helloworld.ServiceAttributes">
 12:          <attribute name="header" value="-> "/>
 13:          <attribute name="count" value="1"/>
 14:       </attributes>
 15:       <controller desc="primitive"/>
 16:    </component>
 17:
 18:    <binding client="this.m" server="client.m"/>
 19:    <binding client="client.s" server="server.s"/>
 20:
```

[1] http://fractal.objectweb.org/tutorials/adl/index.html

```
21:    <controller desc="composite"/>
22:
23:    <virtual-node name="helloworld-node"/>
24:</definition>
25:
```

Now, here is a detailed description of each lines:

- 1: Classical prologue of XML files.
- 2: The syntax of the document is validated against a DTD retrieved from the classpath attribute.
- 4: The **definition** element has a name (which has to be the same name that the file's without its extension). Inheritance is supported through the 'extends' attribute.
- 5: The **interface** element allows to specify interfaces of the current enclosing component.
- 7-16: 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.
- 10: Primitive components specify the **content** element, which indicates the implementation class containing the business logic for this component.
- 11-14: Components can specify a **attributes** element, which allows to initialize attributes of a component.
- 18-19: The **binding** element specifies bindings between interfaces of components and specifying 'this' as the name of the component refers to the current enclosing component.
- 21: The **controller** elements can have the following 'desc' values: 'primitive' or 'composite'.
- 23: The **virtual-node** element offers distributed deployment information. It can be exported and composed in the exported VirtualNodes element.

  The component will be instantiated on the specified virtual node (or the one that is exported). If there are several nodes mapped on the virtual node, the component will be instantiated on one of the nodes of the virtual node.

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

- 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

4.1.2. 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.

This is particularly useful when you want to use a component created by someone else. The programmer who implemented this component may used his own virtual node that you cannot use. Exporting his virtual node, the programmer enables you to redefine this virtual node and thus, you can deploy the component on every virtual node you want. You can also make composition of virtual nodes deterring you from bothering with the definition (in your deployment descriptor) of many virtual nodes. Moreover, a composition of exported virtual nodes also enables to gather two or several components on a same virtual node. This is an essential aspect which provides you with a means to overcome latency bottlenecks when a lot of communications is needed between several 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"/>
```

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. Collocation can be specified in the enclosing component definition (helloworld-distributed-wrappers.fractal):

```
<exportedVirtualNodes>
    <exportedVirtualNode name="VN1">
      <composedFrom>
        <composingVirtualNode component="client" name="client-node"/>
      </composedFrom>
      <composedFrom>
        <composingVirtualNode component="server" name="server-node"/>
      </composedFrom>
    </exportedVirtualNode>
</exportedVirtualNodes>
```

As a result, the client and server component will be collocated / 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 (helloworld-local-no-wrappers). This can be useful for debugging purposes.

```
<exportedVirtualNodes>
    <exportedVirtualNode name="null">
      <composedFrom>
        <composingVirtualNode component="client" name="client-node"/>
      </composedFrom>
    </exportedVirtualNode>
    <exportedVirtualNode name="null">
      <composedFrom>
        <composingVirtualNode component="server" name="server-node"/>
      </composedFrom>
    </exportedVirtualNode>
</exportedVirtualNodes>
```

For more information on exported virtual nodes, please refer to [PhD-Morel]

4.1.3. Usage

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

```
Factory factory = org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory();
Map<String, Object> context = new HashMap<String, Object>();
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, instantiates and assembles the components as defined in the ADL. This launcher is defined in the org.objectweb.proactive.core.component.adl.Launcher class and it can be used as follows:

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

where [-java|-fractal] comes from the Fractal ADL Launcher (put -fractal for ProActive/GCM components), <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.

4.2. Implementation specific API

4.2.1. API and bootstrap component

The API is the same as for any Fractal implementation, though some classes are GCM-specific or implementation-specific.

Thus to get the bootstrap component, there are three possibilities:

- Use the standard Fractal API with one of the methods org.objectweb.fractal.util.Fractal#getBootstrapComponent(...). In that case, the fractal.provider system property has to be set.
- Use the standard GCM API with one of the methods org.etsi.uri.gcm.util.GCM#getBootstrapComponent(...). In that case, the gcm.provider system property has to be set.
- Use the ProActive/GCM API with one of the methods org.objectweb.proactive.core.component.Utils#getBootstrapComponent(...). In that case, the system property to be set can be either gcm.provider or fractal.provider. This last solution is the one used in all examples and tests provided through ProActive.

In all cases, for this implementation the used system property has to be set to org.objectweb.proactive.core.component.Fractive.

The org.objectweb.proactive.core.component.Utils class also contains several useful methods to handle components. Moreover, as for any Fractal or GCM implementation, ProActive/GCM also supports the use of other methods of the org.objectweb.fractal.util.Fractal and org.etsi.uri.gcm.util.GCM classes.

4.2.2. Requirements

As this implementation is based on ProActive, several conditions are required (more in Chapter 2. Active Objects: Creation And Advanced Concepts[2]):

- 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 Chapter 2.7. Asynchronous calls and futures[3]) 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 called method 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.

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/ActiveObjectCreation.html#ActiveObjectCreation

[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/ActiveObjectCreation.html#FutureObjectCreation

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.

4.2.3. Content and controller descriptions

When a component is instantiated with the newFcInstance(Type type, Object controllerDesc, Object contentDesc) 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 targeting a control interface visit the different controllers until they find the suitable one. Then, the request is executed on this controller.

The role of the content description (org.objectweb.proactive.core.component.ContentDescription) is to define some information 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 Chapter 2. Active Objects: Creation And Advanced Concepts[4] for more information about activity in ProActive.
- the meta-object factory for the component (optional).

4.2.4. Collective interfaces

Collective interactions are a GCM extension to the Fractal model, described in section Section 4.3, “Collective interfaces”, that relies on collective interfaces.

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

4.2.5. Monitor controller

By using the org.etsi.uri.gcm.api.control.MonitorController controller, users can retrieve various statistics on components such 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 (Quality of Service) and then decide to do some changes in their application to improve its performance.

After having started the monitoring (Method startGCMMonitoring()), 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.control.MethodStatistics, which itself provides some statistics related to the 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.
 */
```

[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/ActiveObjectCreation.html#ActiveObjectCreation

```
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, Object> getAllGCMStatistics();`

  Which returns the statistics of each methods of each server interfaces of the monitored component through a Map where values are objects which can be cast to MethodStatistics. The key to obtain the MethodStatistics corresponding to a method can be generated thanks to the method org.objectweb.proactive.core.component.control.MonitorControllerHelper#generateKey() which takes as parameters the interface name, the method name and an array of parameter types of the method.
- `public Object getGCMStatistics(String itfName, String methodName, Class<?>[] parametersTypes)`

  Which returns an object which can be cast to MethodStatistics and corresponds to the statistics of 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 get statistics.

4.2.6. 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 available 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.

The following picture represents the addition of non functional requests depending on their priorities:

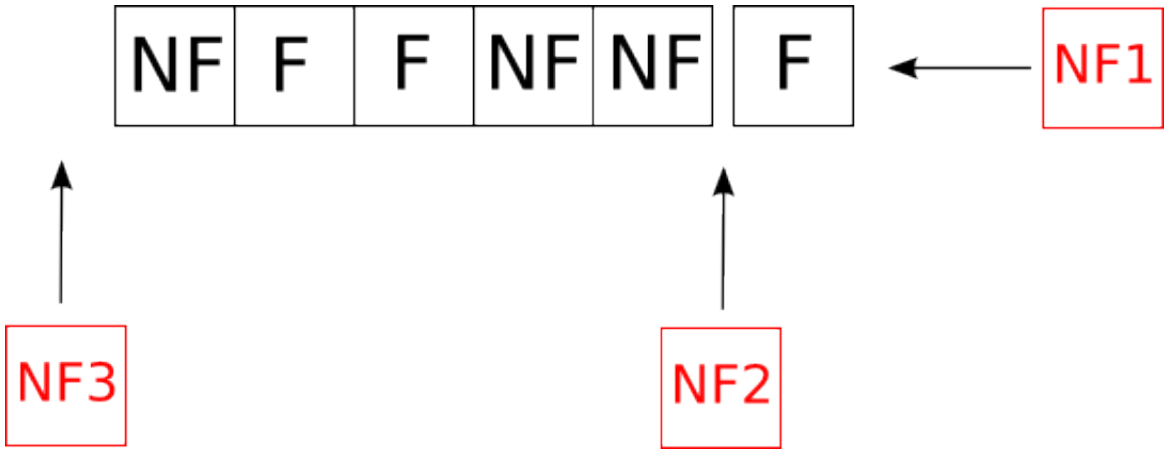


Figure 4.1. Addition of non functional prioritized request

Thus, for prioritize non functional requests, the org.etsi.uri.gcm.api.control.PriorityController controller has to be used:

```
public interface PriorityController {

  /**
   * All the possible kinds of priority for a request on the component to which this interface belongs.
   *
   */
  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 priorities (NF1 and
       * NF2)
       */
      NF3;
  }

  /**
   * Set priority of a method exposed by a server interface of the component to which this interface belongs.
   *
   * @param itfName Name of an interface of the component to which this interface belongs.
   * @param methodName Name of a method exposed by the interface corresponding to the given interface name.
   * @param parameterTypes Parameter types of the method corresponding to the given method name.
   * @param priority Priority to set to the method corresponding to the given method name.
   * @throws NoSuchInterfaceException If there is no such server interface.
   * @throws NoSuchMethodException If there is no such method.
   */
  public void setGCMPriority(String itfName, String methodName, Class<?>[] parameterTypes,
          RequestPriority priority) throws NoSuchInterfaceException, NoSuchMethodException;

  /**
   * Get the priority of a method exposed by a server interface of the component to which this interface
   * belongs.
   *
   * @param itfName Name of an interface of the component to which this interface belongs.
   * @param methodName Name of a method exposed by the interface corresponding to the given interface name.
   * @param parameterTypes Parameter types of the method corresponding to the given method name.
   * @return Priority of this method.
   * @throws NoSuchInterfaceException If there is no such server interface.
   * @throws NoSuchMethodException If there is no such method.
   */
  public RequestPriority getGCMPriority(String itfName, String methodName, Class<?>[] parameterTypes)
```

```
        throws NoSuchInterfaceException, NoSuchMethodException;
}
```

4.2.7. Stream ports

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

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

4.3. 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.

The 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 interface level.

4.3.1. 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.

The GCM proposes 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 multi-way 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

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

4.3.2. Multicast interfaces

4.3.2.1. 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, of the distribution of the invocation parameters and of the result (if there is) are customizable (through annotations).

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** .

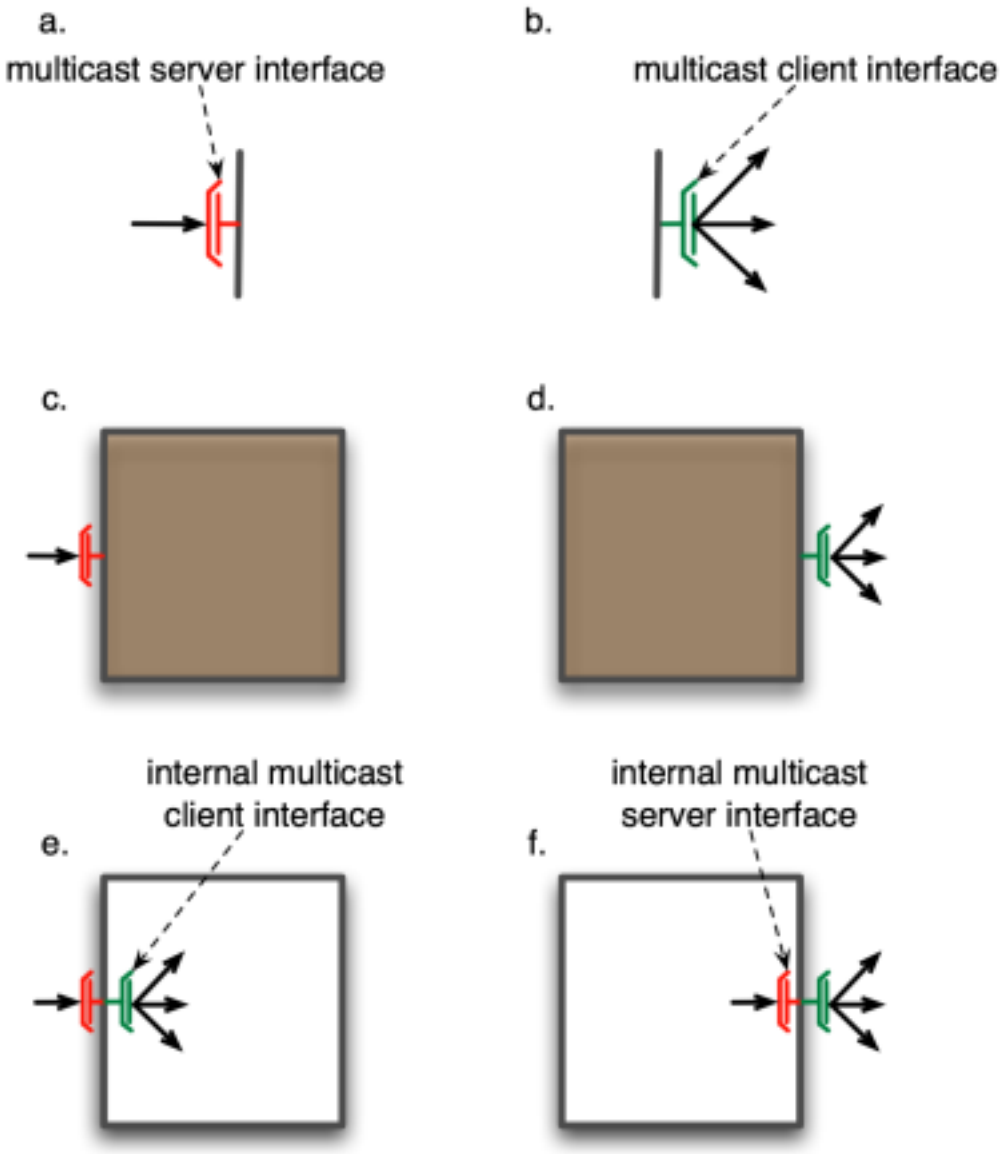


Figure 4.2. Multicast interfaces for primitive and composite components

4.3.2.2. 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 where 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.)

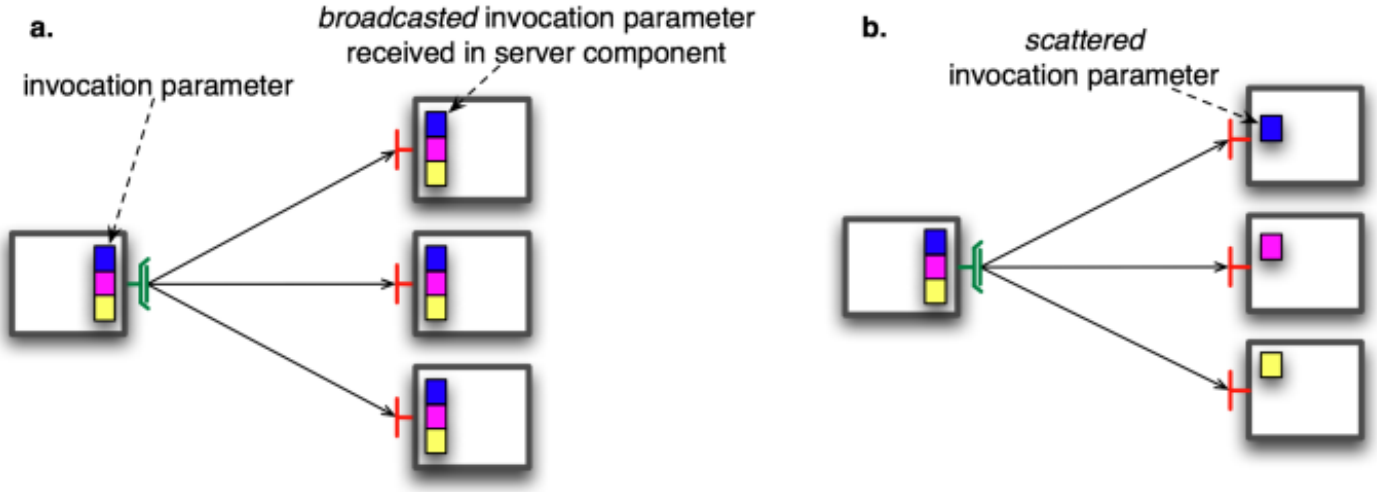


Figure 4.3. Broadcast and scatter of invocation parameters

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

4.3.2.2.1. Invocation parameters distribution modes

Five modes of parameter distribution are provided by default, defining distribution policies for a list of parameters:

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

  ```
  ParamDispatchMode.BROADCAST
  ```

- **ONE-TO-ONE** - sends the i^{th} 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** - distributes each element of the list parameter in a round-robin fashion to the connected server interfaces.

  ```
  ParamDispatchMode.ROUND_ROBIN
  ```

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

  ```
  ParamDispatchMode.RANDOM
  ```

- **UNICAST** - 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))
```

4.3.2.2.2. Configuration through annotations

Note

This 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 this 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:

```
import java.util.List;
import org.objectweb.proactive.examples.documentation.classes.T;

import org.objectweb.proactive.core.component.type.annotations.multicast.ClassDispatchMetadata;

import org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatchMetadata;
import org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatchMode;


@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:

```
import java.util.List;
import org.objectweb.proactive.examples.documentation.classes.T;

import org.objectweb.proactive.core.component.type.annotations.multicast.MethodDispatchMetadata;

import org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatchMetadata;
import org.objectweb.proactive.core.component.type.annotations.multicast.ParamDispatchMode;


interface MyMulticastItf {

   @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 than bound server interfaces. A partitions is a set of arguments (list elements) that will be sent to server interfaces. For instance, instead of giving arguments one-by-one, you can use a 5-size partition which will send the first 5 arguments to the first server interface, the 5 next to the second interface and so on. 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 = 1024)
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 bar(@ParamDispatchMetadata(mode = ParamDispatchMode.BROADCAST)
List<T> parameters);
```

4.3.2.2.3. Results

The previous examples have been given with methods which do not return a result since we were focused on parameters. However, it is obviously possible to use multicast with server interfaces returning non-void results. This section aims at giving explanation on how to handle results.

For each invoked method which returns 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<T> 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 provided by default: **SELECT_UNIQUE_VALUE**. It allows to extract of the list of results the only one result that the list contains. For example, this is useful when your multicast mode is UNICAST, so you know that there will be only one element in your returned list.

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:

```
import java.util.List;

import org.objectweb.proactive.core.component.type.annotations.multicast.Reduce;
import org.objectweb.proactive.core.component.type.annotations.multicast.ReduceMode;
import org.objectweb.proactive.examples.documentation.classes.T;


public interface MyMulticastItf2 {

   @Reduce(reductionMode = ReduceMode.SELECT_UNIQUE_VALUE)
   public T baz();
}
```

Or else, a custom reduce mode can also be used. For this case, the first step is to define 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 = GetLastReduction.class)
public T foobar();
```

This method will use the GetLastReduction algorithm which returns the last element of the list.

Here is the the implementation of the GetLastReduction class:

```
package functionalTests.component.collectiveitf.reduction.composite;

import java.io.Serializable;
import java.util.List;

import org.objectweb.proactive.core.component.exceptions.ReductionException;
import org.objectweb.proactive.core.component.type.annotations.multicast.ReduceBehavior;


public class GetLastReduction implements ReduceBehavior, Serializable {
   public Object reduce(List<?> values) throws ReductionException {
      System.out.println("--------------");
      System.out.println("Getting last out of " + values.size() + " elements");
      System.out.println("--------------");
```

```
        return values.get(values.size() - 1);
    }
}
```

4.3.2.3. Binding compatibility

Multicast interfaces manipulate lists of parameters (sayList<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 picture **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 the picture **b**, the bar method distributes elements of type A to the connected server interfaces.

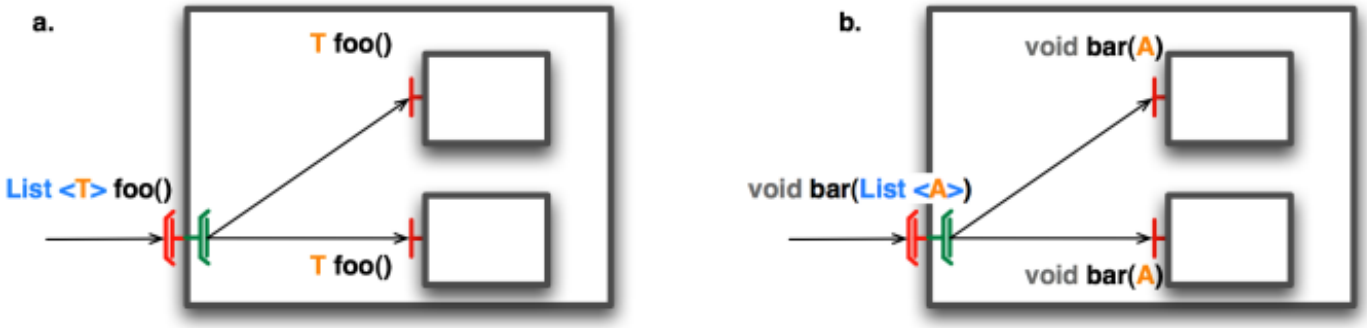


Figure 4.4. Comparison of signatures of methods between client multicast interfaces and server interfaces

For a given multicast interface, the type of server interfaces which may be connected to it can be inferred 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.

4.3.3. Gathercast interfaces

4.3.3.1. 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 on 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.

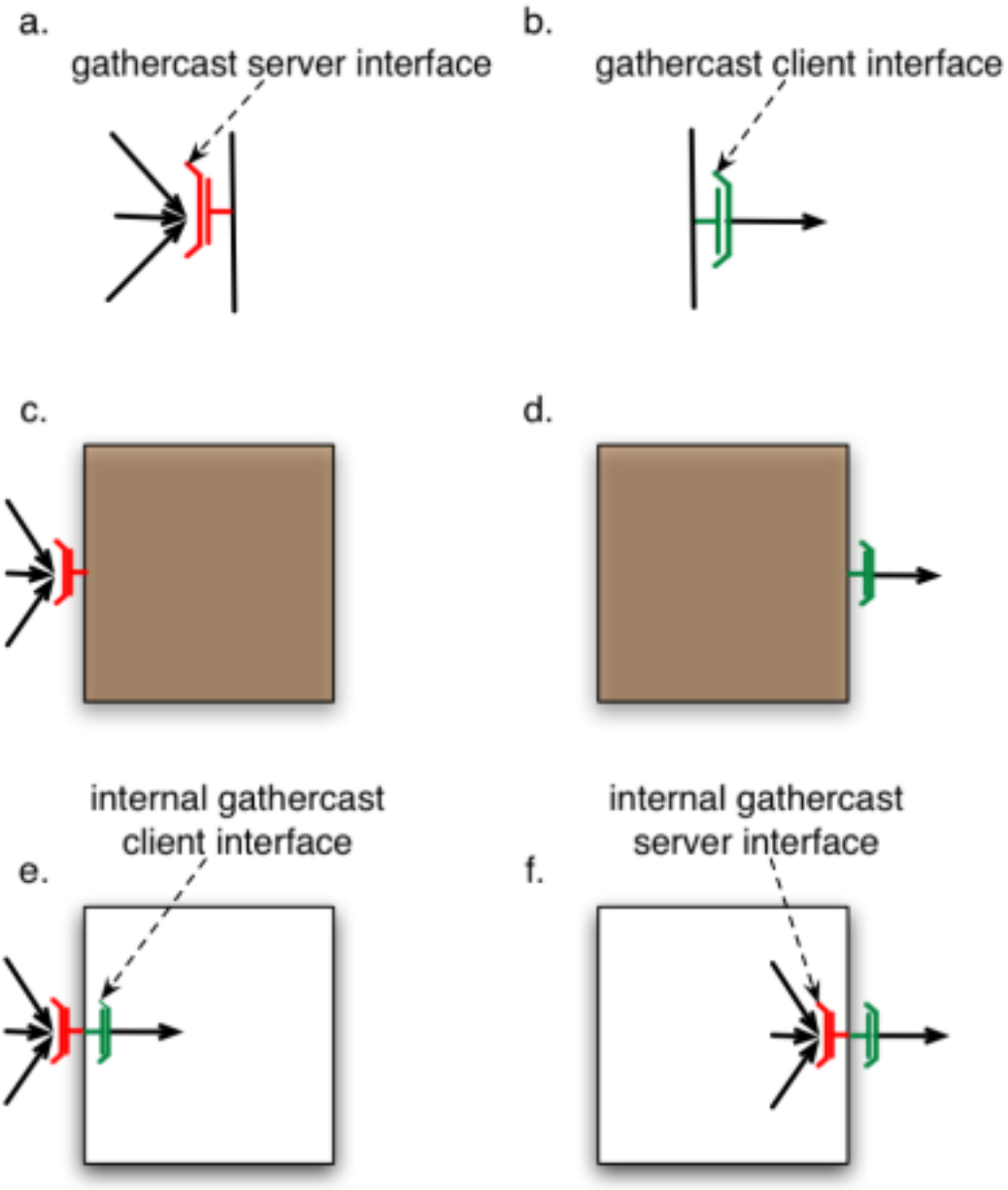


Figure 4.5. Gathercast interfaces for primitive and composite components

4.3.3.2. 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.

4.3.3.2.1. Gathering of invocation parameters

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

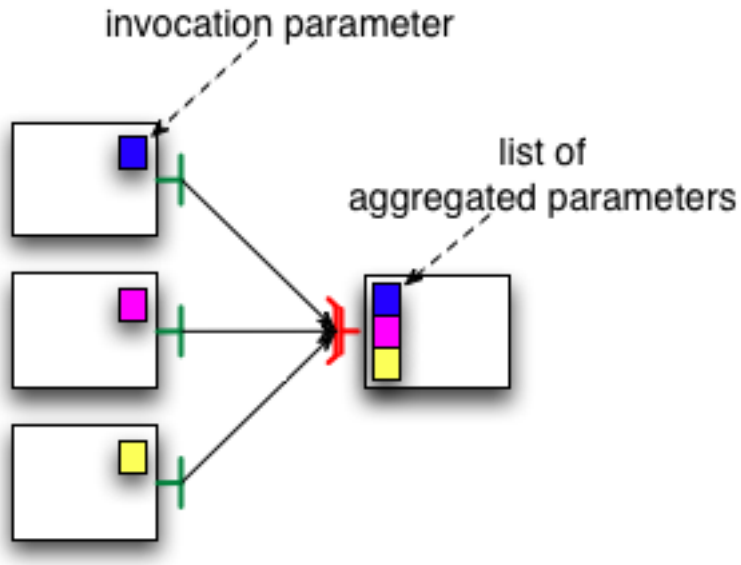


Figure 4.6. Aggregation of parameters with a gathercast interface

4.3.3.2.2. 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.

4.3.3.3. 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 interfaces 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 methods 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 whereas 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.

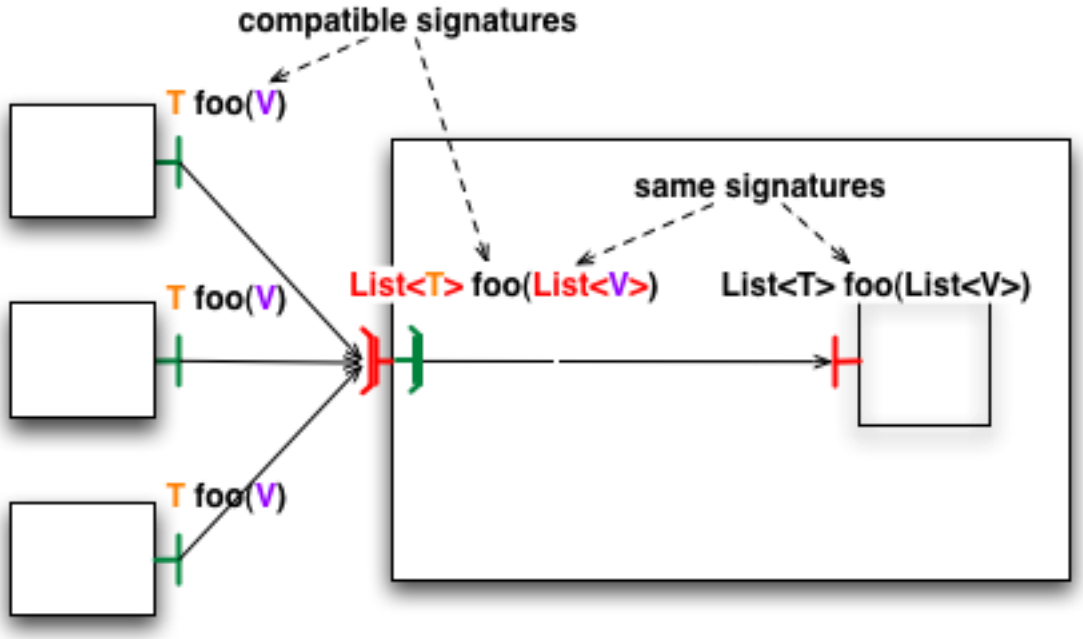


Figure 4.7. Comparison of method signatures for bindings to a gathercast interface

4.3.3.4. Process synchronization

An invocation from a client interface to a gathercast interface is asynchronous, which matches with 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 when the first invocation of a client interface is processed by the gathercast interface, and those when 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 would be raised) because it would be incoherent.

4.4. Deployment

4.4.1. Deploying components with the GCM Deployment

4.4.1.1. 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 14. ProActive Grid Component Model Deployment[5].

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

4.4.1.2. Initiate the deployment

The first step to distribute components is to initiate the deployment by loading the GCM 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.

4.4.1.3. Distribute components with ADL

Distribute components with 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 GCM Application 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 GCM Application Descriptor).

4.4.1.4. Distribute components with API

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

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

[5] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/GCMDeployment.html#GCMDeployment

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

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

4.4.2. Deploying components with the ProActive Deployment

The distribution of components may also be made by using the ProActive deployment framework. This deployment framework uses the concept of virtual nodes too but it just needs a single configuration file. Thus it may be used in a similar way. More information are available at Chapter 15. XML Deployment Descriptors[6].

4.4.3. Deploying components with ADL and Virtual Nodes

It is also possible to deploy components with ADL without using a deployment (GCM or ProActive). To do this, the virtual node has to be directly set in the context using as key the virtual node name:

```
context.put(vn.getName(), vn);
```

Of course, as many virtual nodes as needed can be set in the context. Then the component can be instantiated as usual:

```
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 set in the context).

4.4.4. Deploying components with ADL and Nodes

There is a last way to deploy components with ADL and without using deployment (GCM or ProActive) nor virtual node, using a list of nodes: java.util.List<Node>. This way is mostly useful for tests purpose since there is no control on which node each component will be instantiated.

To deploy components using a list of nodes, the first step is to set this list in the context by setting the "nodes" key (or by using the corresponding constant org.objectweb.proactive.core.component.adl.nodes.ADLNodeProvider.NODES_ID):

```
context.put("nodes", nodes);
```

Then the component can be instantiated as usual:

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

Therefore, the component will be instantiated on a node picked from the list. If the component is a composite component and has several subcomponents, each of them will be instantiated on different nodes. The node used is chosen randomly and there is no way to control which node is used to deploy a specific component. The only guarantee is that no node will be used twice until every node has been used at least once.

4.4.5. Precisions for component instantiation with ADL

1. If no virtual node has been specified in the ADL component definition, the component is created in the local JVM.
2. If a virtual node is defined in the ADL component definition and neither a deployment (GCM or ProActive) nor the virtual node has been set in the context (or there is no corresponding virtual node name), then the component is also created in the local JVM.

Note

These two points do not apply if a list of nodes is set in the context.

[6] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/XML_Descriptors.html

4.5. Advanced

4.5.1. Controllers

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

4.5.1.1. 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../../../../../../Core/org/objectweb/proactive/core/component/config/
component-config.xsd"
  name="MyConfigurationName">
  <controllers>
    <controller>
      <interface>
        ControllerInterface
      </interface>
      <implementation>
        ControllerImplementation
      </implementation>
    </controller>
    <!-- other controllers -->
  </controllers>
</componentConfiguration>
```

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, multicast, gathercast, monitor and priority controllers, the interceptor controller is not available by default).

A custom configuration file can be specified (in this example with "thePathToMyConfigFile") for any component in the controller tag of the ADL definition:

```
<definition name="name">
  ...
  <controller desc="thePathToMyControllerConfigFile"/>
  </definition>
```

Or in the controller description parameter of the newFcInstance method from the Fractal/GCM API:

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

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

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

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

```
protected void setControllerItfType() {
   try {
      setItfType(PAGCMTypeFactoryImpl.instance().createFcItfType("controllername-controller",
            ControllerItf.class.getName(), TypeFactory.SERVER, TypeFactory.MANDATORY,
            TypeFactory.SINGLE));
   } catch (InstantiationException e) {
      throw new ProActiveRuntimeException("cannot create controller type: " + this.getClass().getName());
   }
}
```

4. The controller interface and its implementation have to be declared in the component configuration file.
5. The controller interface name must end by "-controller".

4.5.2. Exporting components as Web Services

4.5.2.1. Overview

This feature allows to expose interfaces of a component as a web service and thus, to call them from any client written in any foreign language.

Indeed, applications written in C#, for instance, cannot communicate with ProActive applications. We have chosen web services technology to enable interoperability because they are based on XML and HTTP. Thus, any component interface can be accessible from any enabled web services language.

4.5.2.2. Principles

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

- **The SOAP Message**

  The SOAP message is used to exchange XML based data over the internet. It can be sent via HTTP and provides a serialization format for communicating over a network.
- **The HTTP Server**

  HTTP is the standard web protocol generally used over the 80 port. In order to receive SOAP messages, you can either install an HTTP server that will be responsible of the data transfer or use the default HTTP server which is a Jetty server. However, This server is not sufficient to treat a SOAP request. For this, you also need a SOAP engine.
- **The SOAP Engine**

  A SOAP Engine is the mechanism responsible of making transparent the unmarshalling of the request and the marshalling of the response. Thus, the service developer doesn't have to worry with SOAP. In our case, we use Apache CXF which can be installed into any HTTP server.

- **The client**

  Client's role is to consume a web service. It is the producer of the SOAP message. The client developer doesn't have to worry about how the service is implemented. The CXF java libraries provide classes to easily invoke a service (see examples).

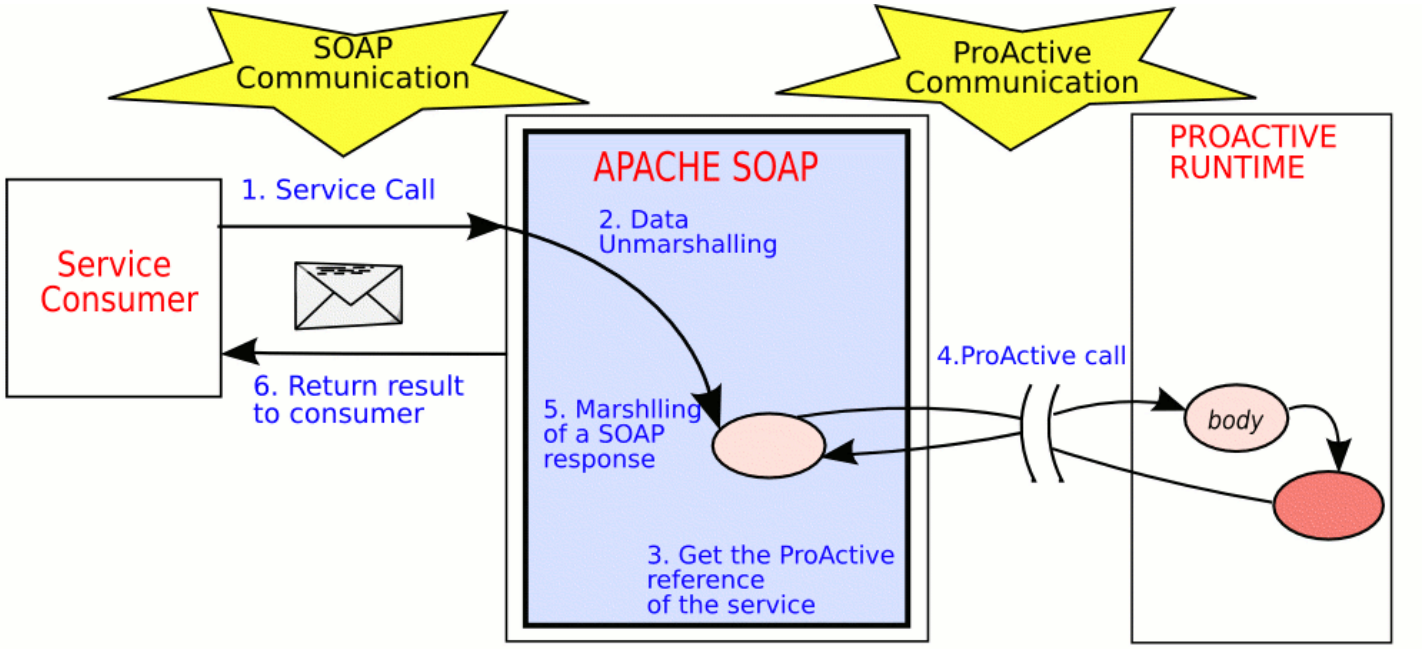


Figure 4.8. Steps taken when an active object is called via SOAP

4.5.2.3. Pre-requisite: Installing the Web Server and the SOAP engine

If you want to expose your components on the local embedded Jetty server or on Jetty servers deployed during your deployment, you just have to deploy ProActive using the build deploy command into the compile directory. If you want to expose them on an other http server, you have to build the proactive.war archive. For this, you have to go into your compile directory of your ProActive home and type build proActiveWar. The proactive.war file will be built into the dist directory.

If you have chosen to use the default Jetty server, then you have nothing else to do. Jetty server will automatically take files it needs.

If you have chosen to use your own HTTP server, then you just have to copy the proactive.war file into the webapp directory of your HTTP server. Some HTTP servers need to be restarted to take into account this new web application but some others like Tomcat can handle hot deployment and, thus, do not need to be restarted.

Warning

If you use your own HTTP server, you have to be aware that the Jetty server will be launched. Most of ProActive examples are launched with the option '-Dproactive.http.port=8080' which specifies Jetty port. So, if you want to use your own server on port 8080, you have to modify the jetty port in launching scripts or to change the port of your own server.

4.5.2.4. Steps to expose/unexpose a component as a web services using the WebServices interface

There are two ways of exposing and unexposing a component as a web service: the first one is the same as we can do with active objects and the second one uses the PAWebServicesController, which is a component controller dedicated to the web services exposition. This section deals with the first method.

The steps for exporting and using a component as a web service are the following:

- Create, instantiate and start your component in a classic way as below:

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

GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
GenericFactory cf = GCM.getGenericFactory(boot);

// type of server component
ComponentType sType = tf.createFcType(new InterfaceType[] { tf.createFcItfType("hello-world", A.class
      .getName(), false, false, false) });
// create server component
Component a = cf.newFcInstance(sType, new ControllerDescription("server", Constants.PRIMITIVE),
      new ContentDescription(AImpl.class.getName()));
//start the component
GCM.getGCMLifeCycleController(a).startFc();
```

- Once the element created and activated, a **WebServicesFactory** object has to be instantiated:

```
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getDefaultWebServicesFactory();
```

or

```
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getWebServicesFactory("cxf");
```

- Then, you have to instantiate a **WebServices** object as follows:

```
WebServices ws = wsf.getWebServices(myUrl);
```

If you want to use the local Jetty server to expose your component, you can use the following line to retrieve the good URL:

```
String myUrl = AbstractWebServicesFactory.getLocalUrl();
```

This will get the Jetty port which is a random port and build the url.

- And finally, you can use one of the WebServices methods:

```
/**
 * Expose a component as a web service. Each server interface of the component
 * will be accessible by  the urn [componentName]_[interfaceName].
 * Only the interfaces public methods of the specified interfaces in
 * <code>interfaceNames</code> will be exposed.
 *
 * @param component The component owning the interfaces that will be deployed as web services.
 * @param componentName Name of the component
 * @param interfaceNames Names of the interfaces we want to deploy.
  *                      If null, then all the interfaces will be deployed
 * @throws WebServicesException
 */
public void exposeComponentAsWebService(Component component, String componentName, String[]
 interfaceNames)
      throws WebServicesException;

/**
 * Expose a component as web service. Each server interface of the component
 * will be accessible by  the urn [componentName]_[interfaceName].
 * All the interfaces public methods of all interfaces will be exposed.
```

```
     *
     * @param component The component owning the interfaces that will be deployed as web services.
     * @param componentName Name of the component
     * @throws WebServicesException
     */
    public void exposeComponentAsWebService(Component component, String componentName)
            throws WebServicesException;

    /**
     * Undeploy all the client interfaces of a component deployed on a web server.
     *
     * @param component  The component owning the services interfaces
     * @param componentName The name of the component
     * @throws WebServicesException
     */
    public void unExposeComponentAsWebService(Component component, String componentName)
            throws WebServicesException;

    /**
     * Undeploy the given client interfaces of a component deployed on a web server.
     * If the array of interface names is null, then undeploy all the interfaces of
     * the component.
     *
     * @param component  The component owning the services interfaces
     * @param componentName The name of the component
     * @param interfaceNames Interfaces tp be undeployed
     * @throws WebServicesException
     */
    public void unExposeComponentAsWebService(Component component, String componentName,
            String[] interfaceNames) throws WebServicesException;

    /**
     * Undeploy specified interfaces of a component deployed on a web server
     *
     * @param componentName The name of the component
     * @param interfaceNames Interfaces to be undeployed
     * @throws WebServicesException
     */
    public void unExposeComponentAsWebService(String componentName, String[] interfaceNames)
            throws WebServicesException;
```

where:

- **component** is the component
- **componentName** the first part of the service name which will identify the component on the server. The service will be exposed at the address http://[host]:[port]/proactive/services/componentName_interfaceName
- **interfaceNames** is a String array containing the interface names of the component you want to make accessible. If this parameters is null (or is absent), all the functional server interfaces of the component will be exposed.

4.5.2.5. Steps to expose/unexpose a component as a web services using PAWebServicesController

As said in the previous section, it is possible to expose web services in two different manners. This section explains how to proceed with the dedicated web services controller.

- First, instantiate your component with the web service controller:

```
// Get the web services controller configuration
String controllersConfigFileLocation = AbstractPAWebServicesControllerImpl
        .getControllerFileUrl("cxf").getPath();

// Create the component using this controller
ControllerDescription cd = new ControllerDescription("server", Constants.PRIMITIVE,
    controllersConfigFileLocation);
Component myComp = genericFactory.newFcInstance(componentType, cd, new ContentDescription(
    HelloWorldComponent.class.getName()));
```

The first line is meant to retrieve the path of the controller file. This file is located at "<your ProActive Home>/src/Extensions/org/objectweb/proactive/extensions/webservices/cxf/initialization/cxf-component-config.xml".

- Then, get the controller to be able to use it:

```
// Get the web services controller
PAWebServicesController wsc = org.objectweb.proactive.extensions.webservices.component.Utils
        .getPAWebServicesController(myComp);
```

- Then, you have to set the url where you want the controller exposes the service:

```
// Set the Url where the component has to be exposed
wsc.setUrl("http://localhost:8080/");
```

If you want to use the local Jetty server to expose your component, you can use the following line to retrieve the good URL:

```
String url = wsc.getLocalUrl();
```

This will get the Jetty port which is a random port and build the url.

- And finally, you can use one of the PAWebServicesController methods:

```
/**
 * Expose a component as a web service. Each server interface of the component
 * will be accessible by  the urn [componentName]_[interfaceName].
 * Only the interfaces public methods of the specified interfaces in
 * <code>interfaceNames</code> will be exposed.
 *
 * @param componentName Name of the component
 * @param interfaceNames Names of the interfaces we want to deploy.
 *                       If null, then all the interfaces will be deployed
 * @throws WebServicesException
 */
public void exposeComponentAsWebService(String componentName, String[] interfaceNames)
        throws WebServicesException;

/**
 * Expose a component as web service. Each server interface of the component
 * will be accessible by  the urn [componentName]_[interfaceName].
 * All the interfaces public methods of all interfaces will be exposed.
 *
 * @param componentName Name of the component
 * @throws WebServicesException
 */
public void exposeComponentAsWebService(String componentName) throws WebServicesException;
```

```
/**
 * Undeploy all the client interfaces of a component deployed on a web server.
 *
 * @param componentName The name of the component
 * @throws WebServicesException
 */
public void unExposeComponentAsWebService(String componentName) throws WebServicesException;

/**
 * Undeploy specified interfaces of a component deployed on a web server
 *
 * @param componentName The name of the component
 * @param interfaceNames Interfaces to be undeployed
 * @throws WebServicesException
 */
public void unExposeComponentAsWebService(String componentName, String[] interfaceNames)
        throws WebServicesException;
```

where:

- **componentName** the first part of the service name which will identify the component on the server. The service will be exposed at the address http://[host]:[port]/proactive/services/componentName_interfaceName
- **interfaceNames** is a String array containing the interface names of the component you want to make accessible. If this parameters is null (or is absent), all the functional server interfaces of the component will be exposed.

4.5.2.6. Exposing as a web service on remote Jetty servers launched during the deployment

When you call the exposition of your component, the CXF servlet is deployed on the local Jetty server. It is therefore possible to expose your object locally without doing anything else.

Now, let us assume that your component is deployed on a remote node. In that case, you have a running jetty server on the remote host containing this node but no servlet has been deployed on it. So, if you want to expose your component on the remote host, you will have to make a small modification of the previous piece of code.

The web service extension provides classes that implements the InitActive interface which are in charge of deploying the CXF servlet on the host where your object has been initialized. Thus, you just have to instantiate your component using this implementation of InitActive.

```
// Get the CXF InitActive in charge of deploying the CXF Servlet
InitActive cxfInitActive = WebServicesInitActiveFactory.getInitActive("cxf");
Component myComponent = ((PAGenericFactory) genericFactory).newFcInstance(componentType,
      new ControllerDescription("myControllerName", Constants.PRIMITIVE), new ContentDescription(
         HelloWorldComponent.class.getName(), null, cxfInitActive, null), myNode);
```

If you have chosen to use the PAWebServicesController controller, you do not need to modified your component instantiation but instead, before the first exposition, you just have to call the following method on the controller:

```
// Deploy the web service servlet on the jetty server corresponding to the node where the component has been
// deployed.
wsc.initServlet();
```

Now, let us assume that your component is deployed on a node or locally and that you want to expose it on a host where another node is deployed. On this host, there is only a running Jetty server without any web services servlet. If you are using the web services controller, it is really straight forward. You just have to put a node as an argument of the previous method:

```
// To expose a component on a remote host where this servlet has not been deployed yet, add
```

```
// the node started on this host as argument. You can put as many nodes as you want.
wsc.initServlet(myNode);
```

If you are not using the controller, you can use the following line:

```
WebServicesInitActiveFactory.getInitActive("cxf").initServlet(myNode);
```

Then, you can expose your component as a web service as in Section 4.5.2.4, “Steps to expose/unexpose a component as a web services using the WebServices interface”.

4.5.2.7. Accessing the services

Once the component interface exposed, you can access it via any web service client (such as C#, SoapUI, Apache CXF API, Apache Axis2 API, ...) or with your favorite browser (only if the returned type of your service methods are primitive types).

First of all, the client will get the WSDL file matching this component interface. This WSDL file is the 'identity card' of the service. It contains the web service public interfaces and its location. Generally, WSDL files are used to generate a proxy to the service. For example, for a given service, say 'componentTest_computeItf', you can get the WSDL document at http://[host]:[port]/proactive/services/componentTest_computeItf?wsdl.

Now that this client knows what and where to call the service, it will send a SOAP message to the web server. The web server looks into the message and performs the right call. Then, it returns the response into another SOAP message to the client.

4.5.2.8. Limitations

Contrary to the previous version of the component exposition using Apache SOAP, this new version which uses Axis2 or CXF can handle complex types. That is, with this version, you can implement a service which exposes a method returning or taking in argument an instance of any class. By using CXF and JAX-WS annotations, there is no limitation on the type of this class. It only implies to use the appropriate adapters (see the CXF documentation [7] for more details). However, if you do not want to use JAX-WS annotations or CXF, the class of this instance has to respect some criterion due to the serialization and Axis2/CXF restrictions:

- This class has to be serializable
- This class has to supply a no-args constructor and preferably empty
- All the fields of this class has to be private or protected
- This class has to supply public getters and setters for each field. These getter and setter methods have to be in this class and not in one of its super classes.

4.5.2.9. A simple example: Hello World

Let's start with a simple example, an Helloworld component with two interfaces exposed as a web service:

First, below are the two server interfaces of the component:

```
public interface HelloWorldItf {

  public String helloWorld(String arg0);

  public String sayHello();

  public String sayText();

  public void setText(String arg0);
```

[7] http://cxf.apache.org/docs/index.html

```
}

public interface GoodByeWorldItf {

   public String goodByeWorld(String arg0);

   public String sayGoodBye();

   public void setText(String arg0);

   public String sayText();
}
```

The implementation class of the component is the following one:

```
public class HelloWorldComponent implements HelloWorldItf, GoodByeWorldItf, Serializable {

   private String text;

   public String sayText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }

   public HelloWorldComponent() {
   }

   public String helloWorld(String arg0) {
      return "Hello " + arg0 + " !";
   }

   public String goodByeWorld(String arg0) {
      return "Good Bye " + arg0 + " !";
   }

   public String sayHello() {
      return "Hello ProActive Team !";
   }

   public String sayGoodBye() {
      return "Good bye ProActive Team !";
   }

}
```

The following piece of code creates the component and exposes its interfaces as a web services.

```
Component componentBoot = Utils.getBootstrapComponent();

GCMTypeFactory typeFactory = GCM.getGCMTypeFactory(componentBoot);
```

```
GenericFactory genericFactory = GCM.getGenericFactory(componentBoot);

// type of server component
ComponentType componentType = typeFactory.createFcType(new InterfaceType[] {
        typeFactory
                .createFcItfType("hello-world", HelloWorldItf.class.getName(), false, false, false),
        typeFactory.createFcItfType("goodbye-world", GoodByeWorldItf.class.getName(), false, false,
                false) });

// create server component
Component helloWorld = genericFactory.newFcInstance(componentType, new ControllerDescription(
    "server", Constants.PRIMITIVE), new ContentDescription(HelloWorldComponent.class.getName()));
//start the component
GCM.getGCMLifeCycleController(helloWorld).startFc();

// As only one framework is supported for the moment, use
// the following method is equivalent to
// WebServicesFactory.getDefaultWebServicesFactory()
WebServicesFactory webServicesFactory = AbstractWebServicesFactory.getWebServicesFactory("cxf");

// If you want to use the local Jetty server, you can use
// AbstractWebServicesFactory.getLocalUrl() to get its url with
// its port number (which is random except if you have set the
// proactive.http.port variable)
WebServices webservices = webServicesFactory.getWebServices("http://localhost:8080/");

// If you want to expose only the goodbye-world interface for example,
// use the following line instead:
// webservices.exposeComponentAsWebService(helloWorld, "MyHelloWorldComponentService", new String[] { "goodbye-
world" });
webservices.exposeComponentAsWebService(helloWorld, "MyHelloWorldComponentService");
```

If you want to use the web service controller, you should write the following line:

```
Component componentBoot = Utils.getBootstrapComponent();

GCMTypeFactory typeFactory = GCM.getGCMTypeFactory(componentBoot);
GenericFactory genericFactory = GCM.getGenericFactory(componentBoot);

// type of server component
ComponentType componentType = typeFactory.createFcType(new InterfaceType[] {
        typeFactory
                .createFcItfType("hello-world", HelloWorldItf.class.getName(), false, false, false),
        typeFactory.createFcItfType("goodbye-world", GoodByeWorldItf.class.getName(), false, false,
                false) });

// Get the web services controller configuration
String controllerPath = AbstractPAWebServicesControllerImpl.getControllerFileUrl("cxf").getPath();

// Create the component using this controller
ControllerDescription controllerDesc = new ControllerDescription("server", Constants.PRIMITIVE,
    controllerPath);

// Create server component
```

```
Component helloWorldComponent = genericFactory.newFcInstance(componentType, controllerDesc,
        new ContentDescription(HelloWorldComponent.class.getName()));

// Start the component
GCM.getGCMLifeCycleController(helloWorldComponent).startFc();

// Get the web services controller
PAWebServicesController wsController = org.objectweb.proactive.extensions.webservices.component.Utils
        .getPAWebServicesController(helloWorldComponent);

// Set the Url where the component has to be exposed
wsController.setUrl("http://localhost:8080/");

// Expose the component as a web service
wsController.exposeComponentAsWebService("MyHelloWorldComponentService");
```

All the interfaces (hello-world and goodbye-world) of the component helloWorld have been deployed as web services on the web server located at http://localhost:8080/proactive/services/ and its service names are "MyHelloWorldComponentService_hello-world" and "MyHelloWorldComponentService_goodbye-world". You can see the wsdl files through http://localhost:8080/proactive/services/MyHelloWorldComponentService_hello-world?wsdl and http://localhost:8080/proactive/services/MyHelloWorldComponentService_goddbye-world?wsdl and you can call the sayHello method of the hello-world interface with your browser through http://localhost:8080/proactive/services/MyHelloWorldComponentService_hello-world/sayHello.

If you have exposed a method which requires arguments, you can call it using the same link as previously but adding ?[arg0]=[value]&[arg1]=[value]... where [argn] is the name of the n-th argument as written in the WSDL file.

You can also call a webservice using the ProActive client which uses a CXF client. Here is an example:

```
// Instead of using "cxf", you can also use webServicesFactory.getFrameWorkId().
// As only one framework is supported for the moment, use the following method
// is equivalent to ClientFactory.getDefaultClientFactory()
ClientFactory cFactory = AbstractClientFactory.getClientFactory("cxf");

// Instead of using "http://localhost:8080/", you can use ws.getUrl() to
// ensure to get to good service address.
Client serviceClient = cFactory.getClient("http://localhost:8080/",
        "MyHelloWorldComponentService_goodbye-world", GoodByeWorldItf.class);

// Call which returns a result
Object[] result = serviceClient.call("goodByeWorld", new Object[] { "ProActive Team" }, String.class);
System.out.println((String) result[0]);

// Call which does not return a result
serviceClient.oneWayCall("setText", new Object[] { "Hi ProActive Team!" });

// Call with no argument
result = serviceClient.call("sayText", null, String.class);

System.out.println((String) result[0]);
```

4.5.3. Web service bindings

ProActive also gives the possibility to bind a functional client interface of a component to a web service.

For instance, this feature is useful to enable communication between SCA (Service Component Architecture) components and GCM/ProActive components by using the web service communication protocol. Effectively, communication from SCA component to GCM/ProActive component can be done thanks to the ability of exposing GCM server interface as web service (See Section 4.5.2, “Exporting components as Web Services”) and communication from GCM/ProActive component to SCA component can be done by using this feature. This has already been successfully tested with two SCA implementations: Apache Tuscany [8] and FraSCAti [9].

The only requirement for binding a client interface to a web service is that the web service must be known in advance in order to ensure the compatibility between the client interface and the web service to bind to. In other words, the Java interface representing the client interface must have the same method signatures than those exposed by the web service.

It is the same principle to bind a client interface to a web service with both the ADL and the API: the web service binding can be done just by replacing the server interface name by the URL of the web service (not the WSDL address) in the call to the method bindFc with the API or in the tag binding with the ADL.

By default, the CXF API is used to call the web service, but it is also possible to specify another library. If so, the URL has to be followed, in parenthesis, by the ID or the full name of the class to use to call the web service. Thus, ProActive offers several ways to call web services by using various configurations of CXF API. IDs are defined in the class org.objectweb.proactive.core.component.webservices.WSInfo. Otherwise, the full class name has to be the one of a class implementing the org.objectweb.proactive.core.component.webservices.PAWSCaller interface:

```
public interface PAWSCaller {
   /**
    * Method to setup the caller.
    *
    * @param serviceClass Class that the web service implements. Should match with
    * the client interface to bind to the web service.
    * @param wsUrl URL of the web service (not the WSDL address).
    */
   public void setup(Class<?> serviceClass, String wsUrl);

   /**
    * Method to call a web service.
    *
    * @param methodName Name of the service to call.
    * @param args Parameters of the web service.
    * @param returnType Class of the return type of the web service. Null if the web
    * service does not return any result.
    * @return Result of the call to the web service if there is, null otherwise or if the
    * invocation failed.
    */
   public Object callWS(String methodName, Object[] args, Class<?> returnType);
}
```

For instance, the replacing string of the server interface name may be:

- "http://localhost:8080/proactive/services/Server_HelloWorld(org.objectweb.proactive.core.component.webservices.CXFWSCaller)"
- which is equivalent to: "http://localhost:8080/proactive/services/Server_HelloWorld(" + org.objectweb.proactive.core.component.webservices.WSInfo.CXFWSCALLER_ID + ")"
- which is also equivalent to: "http://localhost:8080/proactive/services/Server_HelloWorld" since CXF is used by default.

4.5.4. Lifecycle: encapsulation of functional activity in component lifecycle

[8] http://tuscany.apache.org/
[9] https://wiki.ow2.org/frascati/

In this implementation of the Fractal/GCM component model, Fractal/GCM 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 has to use the org.objectweb.proactive.Service class
2. the functional activity has to 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.

Note

By the way, it should be specified that in this implementation of the org.etsi.uri.gcm.api.control.GCMLifeCycleController, the startFc and stopFc methods are recursive (ie. starting or stopping a component, starts or stops all its subcomponents) whereas the terminateGCMComponent method is not recursive (ie. only terminates the current component).

4.5.5. Structuring the membrane with non-functional components

Components running in dynamically changing execution environments need to adapt to these environments. In Fractal/GCM component model, 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. It 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, we aim at a component platform appropriate for **autonomic Grid applications**; those applications aim at ensuring 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; secondly, a definition and an implementation of an API that allows 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 [MB01, SPC01]. 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 the NF part of the 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 model.

4.5.5.1. Motivating example

Here we present a simple example that shows the advantages of componentizing controllers of components. In our example, we are considering a naive solution for securing communications of a composite component. As described in Figure 4.9, "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 in charge of deciding 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.

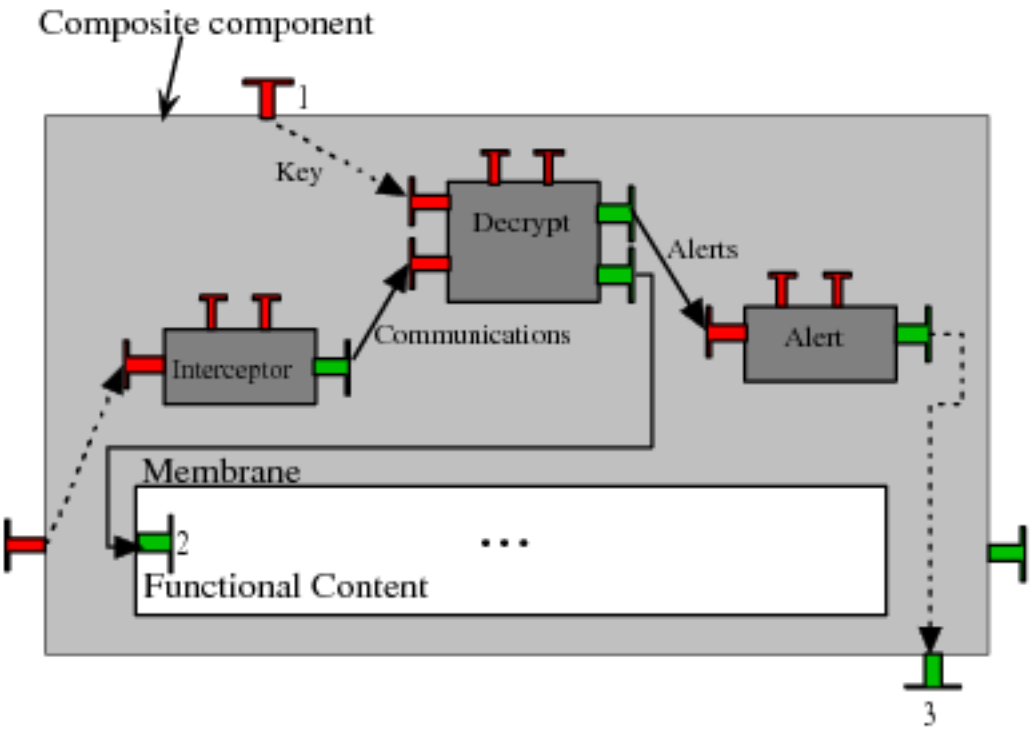


Figure 4.9. Example: architecture of a naive solution for secure communications

4.5.5.2. A structure for Componentized Membranes

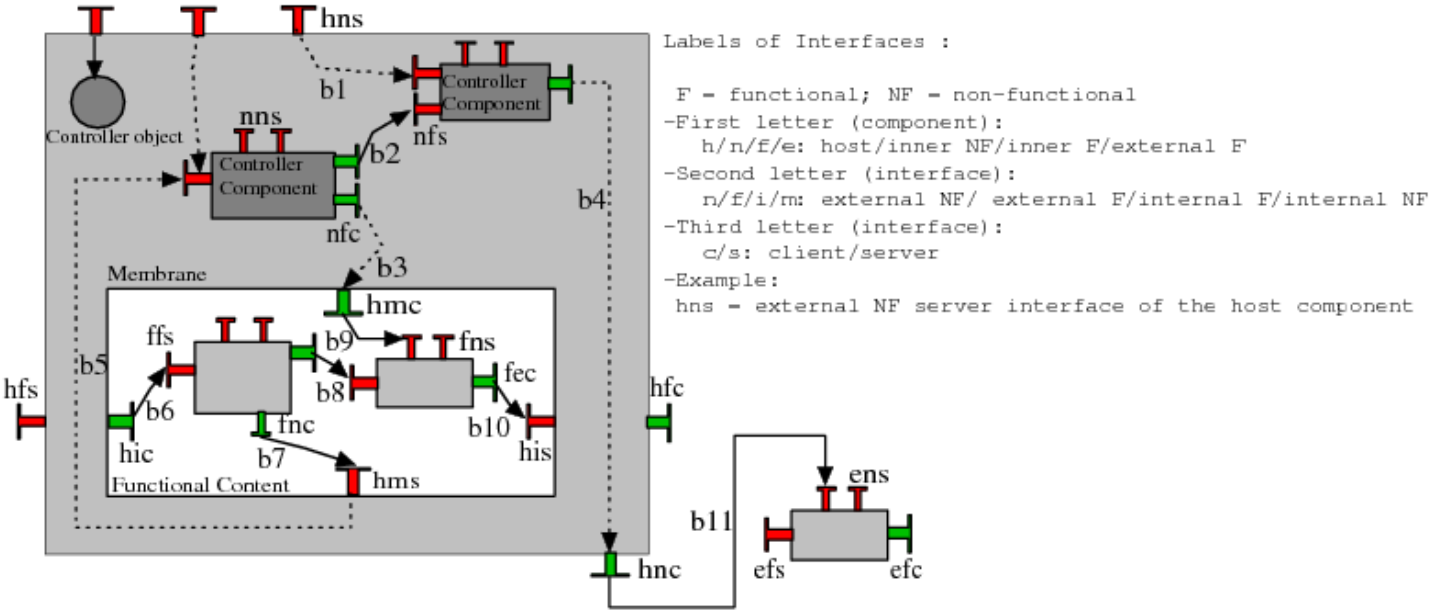


Figure 4.10. Structure for the membrane of Fractal/GCM components

Figure 4.10, “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 4.10, “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 that a 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 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 4.10, “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 4.10, “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.

4.5.5.2.1. 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.

4.5.5.3. An API for (Re)configuring Non-Functional Aspects

4.5.5.3.1. Non-functional Type and Non-functional Components

To type check bindings between membranes, we have to extend the GCM model: the type of a component must includes a non-functional part. This part is defined as the union of the types of NF interfaces the membrane exposes. To specify the NF part in the type of a component, we defined the classes:

- PAComponentType which is an extension of the ComponentType class and adds methods to access to the non functional interface types of a component type.
- PAGCMTypeFactory which is an extension of the GCMTypeFactory class and adds a method to create a PAComponentType.

Then the component can be instantiated as usually by using the newFcInstance method where the type given as parameter is an instance of PAComponentType.

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. In the generic factory, a method named newNfFcInstance that creates this new kind of components has been added:

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

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

```
/**
 * Adds non-functional components inside the membrane.
 *
 * @param component The non-functional component to add.
 * @throws IllegalContentException If the component to add is not a non-functional component.
 */
void nfAddFcSubComponent(Component component) throws IllegalContentException, IllegalLifeCycleException;

/**
 * Removes the specified component from the membrane.
 *
 * @param componentname The name of the component to remove.
 * @throws IllegalContentException If the specified component can not be removed.
 */
void nfRemoveFcSubComponent(Component componentname) throws IllegalContentException,
        IllegalLifeCycleException, NoSuchComponentException;

/**
 * Returns an array containing all the components inside the membrane.
 *
 * @return All the non-functional components.
 */
Component[] nfGetFcSubComponents();

/**
 * Returns the non functional component specified by the name parameter.
 *
 * @return The non functional component specified by the name parameter.
 */
Component nfGetFcSubComponent(String name) throws NoSuchComponentException;

/**
 * Sets a new controller object implementing the specified interface.
 *
 * @param itf The name of interface the new object has to implement.
 * @param controllerclass The controller object.
 * @throws NoSuchInterfaceException If the specified interface does not exist.
 */
void setControllerObject(String itf, Object controllerclass) throws NoSuchInterfaceException;

/**
 * Starts all non functional components inside the membrane.
 *
 * @throws IllegalLifeCycleException If one of the non functional components is in inconsistent lifecycle state.
 */
public void startMembrane() throws IllegalLifeCycleException;

/**
 * Stops all non functional components inside the membrane.
 *
 * @throws IllegalLifeCycleException If one of the non functional components is in inconsistent lifecycle state.
 */
public void stopMembrane() throws NoSuchInterfaceException, IllegalLifeCycleException;

/**
 * Returns the state of the membrane (by default, started or stopped).
 *
 * @return The current state of the membrane.
 */
public String getMembraneState();
```

Figure 4.11. 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/GCM 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 4.11, "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 object 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:

- nfAddFcSubComponent(Component component): adds the NF component given as argument to the membrane;
- nfRemoveFcSubComponent(Component component): removes the specified NF component from the membrane;
- nfGetFcSubComponents(): returns an array containing all the NF components;
- nfGetFcSubComponent(string name): returns the specified NF component, the string argument is the name of the NF component;
- setControllerObject(string itf, any controller class): 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 has 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 has a recursive behavior, by stopping the life-cycle of each NF component.

```
 * @throws IllegalLifeCycleException If one of the component is in inconsistent lifecycle state.
 * @throws IllegalBindingException If the type of the interfaces does not match.
 */
void nfBindFc(String clientItf, String serverItf) throws NoSuchInterfaceException,
    IllegalLifeCycleException, IllegalBindingException, NoSuchComponentException;

/**
 * Performs bindings with non-functional external interfaces.
 *
 * @param clientItf The client interface, referenced by a string of the form "component.interface", where
 * component is the name of the component, and interface the name of its client interface. If the component name
 * is "membrane", it means that interface must be the name of a non-functional external interface of the host
 * component.
 * @param serverItf The non-functional external server interface.
 * @throws NoSuchInterfaceException If one of the specified client interface does not exist.
 * @throws IllegalLifeCycleException If one of the component is in inconsistent lifecycle state.
 * @throws IllegalBindingException If the type of the interfaces does not match.
 */
void nfBindFc(String clientItf, Object serverItf) throws NoSuchInterfaceException,
    IllegalLifeCycleException, IllegalBindingException, NoSuchComponentException;

/**
 * Removes non-functional bindings.
 *
 * @param clientItf The client interface that will have its binding removed. It is a string of the form
 * "component.interface", where component is the name of the component, and interface the name of its server
 * interface. If the component name is "membrane", it means that interface must be the name of a non-functional
 * external/internal interface of the host component.
 * @throws NoSuchInterfaceException If the client interface does not exist.
 * @throws IllegalLifeCycleException If the component is in inconsistent lifecycle state.
 * @throws IllegalBindingException If the binding can not be removed.
 */
void nfUnbindFc(String clientItf) throws NoSuchInterfaceException, IllegalLifeCycleException,
    IllegalBindingException, NoSuchComponentException;

/**
 * Returns the names of the client interfaces belonging to the component, which name is passed as argument.
 *
 * @param component The name of the component.
 * @return An array containing the name of the client interfaces(we suppose that non-functional components don't
 * have client NF interfaces).
 */
String[] nfListFc(String component) throws NoSuchComponentException, NoSuchInterfaceException,
    IllegalLifeCycleException;

/**
 * Returns the stub and proxy of the server interface the client interface is connected to (for components
 * inside the membrane).
 *
 * @param itfname The client interface, referenced by a string of the form "component.interface", where
 * component is the name of the component, and interface the name of its client interface.
 * @return The stubs and the proxy of the server interface the client interface is connected to.
 * @throws NoSuchInterfaceException If the specified interface does not exist.
 */
Object nfLookupFc(String itfname) throws NoSuchInterfaceException, NoSuchComponentException;

/**
 * Starts the specified component.
 *
 * @param component The name of the component to start.
 * @throws IllegalLifeCycleException If the lifecycle state is inconsistent.
 */
void nfStartFc(String component) throws IllegalLifeCycleException, NoSuchComponentException,
    NoSuchInterfaceException;

/**
```

Figure 4.12. Higher level API

In Figure 4.12, “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 do not 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 membrane controller of the host component.

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

Using the API of Figure 4.12, “Higher level API”, this binding can be realized by the following code, that binds the component “nfComp1” correctly.

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

Similarly to the example above, all the methods of Figure 4.12, “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 “membrane” to represent the membrane of the host component, e.g. membrane.i1 is the NF interface i1 of the host component. For example, nfBindFc(string, string) allows to perform the bindings: **b1**, **b2**, **b4**, **b3**, **b9**, **b7** and **b5** of Figure 4.10, “Structure for the membrane of Fractal/GCM components”.

The API presented in Figure 4.12, “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 references 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 nfGetFcSubComponent(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 4.12, “Higher level API” solve this problem.

4.5.6. Short cuts

4.5.6.1. 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.

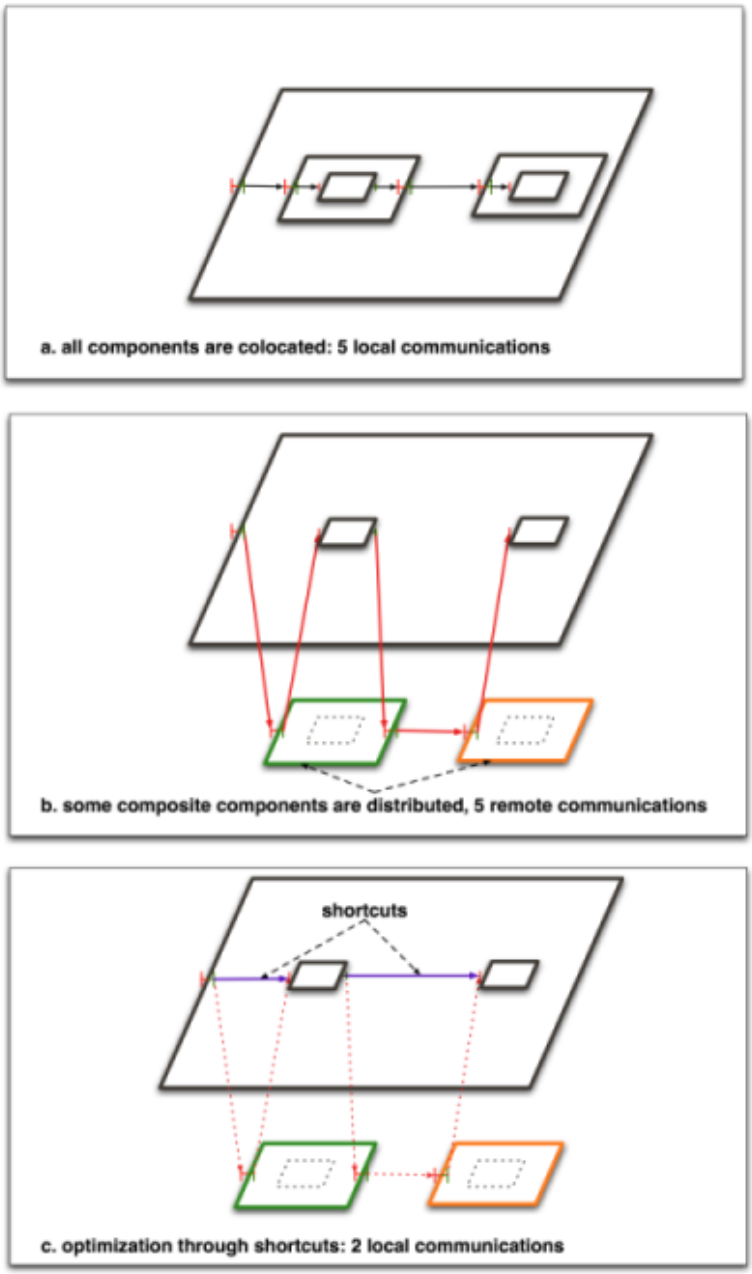


Figure 4.13. Using shortcuts for minimizing remote communications

4.5.6.2. 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.

4.5.7. Interceptors

This section explains how to create and use interceptors.

4.5.7.1. Configuration of interceptors

Controllers and non functional components can also act as interceptors: they can intercept incoming invocations and/or outgoing invocations. To do so, controllers must implement the Interceptor interface and non functional components must provide a functional server interface implementing the Interceptor interface.

For each invocation, pre and post processings are defined in the methods beforeMethodInvocation and afterMethodInvocation. These methods are defined in the Interceptor interface and take an InterceptedRequest object as parameter. An InterceptedRequest object contains the names of the interface and the method on which the intercepted request has been invoked, the parameter and return types of the invoked method and the parameter values of the invoked method. An InterceptedRequest object may also contain the return value of the method invocation if the interception occurs after the execution of the method (ie. in the afterMethodInvocation method). If the method has no return type (ie. returns void), then null is returned as result by the InterceptedRequest object. If the method returns a primitive type, then it is wrapped inside the appropriate wrapper object. beforeMethodInvocation and afterMethodInvocation methods must also return the InterceptedRequest object that has been given as parameter. Thus it allows to the interceptor implementation to modify some information contained into the InterceptedRequest object and to propagate these modifications. Typically, it allows to beforeMethodInvocation implementations to modify the values of some of the parameters before serving the request. In a similar way, it allows to afterMethodInvocation implementations to modify the value of the result of the invocation of the request before returning it. Finally, beforeMethodInvocation and afterMethodInvocation methods may also prevent the normal process of the invocation of the request by throwing a RuntimeException, or any of its subclasses. In such a case, the invocation of the request will be stopped and the exception will be returned to the invoker in case of non one-way request.

Interceptors are configured with the interceptor controller. This controller is not included in the default controllers configuration so it must be explicitely added:

```
<controller>
   <interface>org.objectweb.proactive.core.component.control.PAInterceptorController</interface>
   <implementation>org.objectweb.proactive.core.component.control.PAInterceptorControllerImpl</implementation>
</controller>
```

The management of the interceptors is then done through the methods of the interceptor controller:

```
/**
 * Returns the IDs of the {@link Interceptor interceptors} attached to the interface with the specified name.
 *
 * @param interfaceName Name of the interface on which to get the IDs of its {@link Interceptor interceptors}.
 * @return The IDs of the {@link Interceptor interceptors} attached to the interface with the specified name.
 * @throws NoSuchInterfaceException If there is no such interface.
 */
public List<String> getInterceptorIDsFromInterface(String interfaceName) throws NoSuchInterfaceException;

/**
 * Adds at the specified position the {@link Interceptor interceptor} with the specified ID to the interface
 * with the specified name.
 *
 * @param interfaceName Name of the interface on which to add the {@link Interceptor interceptor}.
 * @param interceptorID ID of the {@link Interceptor interceptor} to add.
 * @param index Position on which to add the {@link Interceptor interceptor}.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface or if the ID does not represent a
 * controller interface nor a server interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
```

```
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor} or if the index is invalid.
 */
public void addInterceptorOnInterface(String interfaceName, String interceptorID, int index)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Adds at the last position the {@link Interceptor interceptor} with the specified ID to the interface
 * with the specified name.
 *
 * @param interfaceName Name of the interface on which to add the {@link Interceptor interceptor}.
 * @param interceptorID ID of the {@link Interceptor interceptor} to add.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface or if the ID does not represent a
 * controller interface nor a server interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void addInterceptorOnInterface(String interfaceName, String interceptorID)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Adds the {@link Interceptor interceptors} with the specified IDs to the interface with the specified
 * name.
 *
 * @param interfaceName Name of the interface on which to add the {@link Interceptor interceptor}.
 * @param interceptorIDs IDs of the {@link Interceptor interceptors} to add.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface or if one of the ID does not represent a
 * controller interface nor a server interface of a NF component.
 * @throws NoSuchComponentException If one the IDs matches the format of the concatenation of a NF component
 * name and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void addInterceptorsOnInterface(String interfaceName, List<String> interceptorIDs)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Adds at the last position the {@link Interceptor interceptor} with the specified ID to all the
 * server interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to add.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
```

```
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void addInterceptorOnAllServerInterfaces(String interceptorID) throws IllegalLifeCycleException,
        NoSuchInterfaceException, NoSuchComponentException, IllegalInterceptorException;

/**
 * Adds at the last position the {@link Interceptor interceptor} with the specified ID to all the
 * client interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to add.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void addInterceptorOnAllClientInterfaces(String interceptorID) throws IllegalLifeCycleException,
        NoSuchInterfaceException, NoSuchComponentException, IllegalInterceptorException;

/**
 * Adds at the last position the {@link Interceptor interceptor} with the specified ID to all the
 * client and server interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to add.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void addInterceptorOnAllInterfaces(String interceptorID) throws IllegalLifeCycleException,
        NoSuchInterfaceException, NoSuchComponentException, IllegalInterceptorException;

/**
 * Removes the {@link Interceptor interceptor} located at the specified position from the interface
 * with the specified name.
 *
 * @param interfaceName Name of the interface on which to remove the {@link Interceptor interceptor}.
 * @param index Position of the {@link Interceptor interceptor} to remove.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface.
 * @throws IllegalInterceptorException If the index is invalid.
 */
public void removeInterceptorFromInterface(String interfaceName, int index)
        throws IllegalLifeCycleException, NoSuchInterfaceException, IllegalInterceptorException;

/**
 * Removes the first occurrence of the {@link Interceptor interceptor} with the specified ID from the
 * interface with the specified name.
```

```
 *
 * @param interfaceName Name of the interface on which to remove the {@link Interceptor interceptor}.
 * @param interceptorID ID of the {@link Interceptor interceptor} to remove.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface or if the ID does not represent a
 * controller interface nor a server interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void removeInterceptorFromInterface(String interfaceName, String interceptorID)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Removes all the {@link Interceptor interceptors} from the interface with the specified name.
 *
 * @param interfaceName Name of the interface on which to remove the {@link Interceptor interceptor}.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If there is no such interface.
 */
public void removeAllInterceptorsFromInterface(String interfaceName) throws IllegalLifeCycleException,
        NoSuchInterfaceException;

/**
 * Removes the first occurrence of the {@link Interceptor interceptor} with the specified ID from all the
 * server interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to remove.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void removeInterceptorFromAllServerInterfaces(String interceptorID)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Removes the first occurrence of the {@link Interceptor interceptor} with the specified ID from all the
 * client interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to remove.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
```

```
 */
public void removeInterceptorFromAllClientInterfaces(String interceptorID)
        throws IllegalLifeCycleException, NoSuchInterfaceException, NoSuchComponentException,
        IllegalInterceptorException;

/**
 * Removes the first occurrence of the {@link Interceptor interceptor} with the specified ID from all the
 * client and server interfaces.
 *
 * @param interceptorID ID of the {@link Interceptor interceptor} to remove.
 * @throws IllegalLifeCycleException If the component is not in the stopped state.
 * @throws NoSuchInterfaceException If the ID does not represent a controller interface nor a server
 * interface of a NF component.
 * @throws NoSuchComponentException If the ID matches the format of the concatenation of a NF component name
 * and a server interface name but there is no NF component with such a name.
 * @throws IllegalInterceptorException If there is a problem with the ID or with the
 * {@link Interceptor interceptor}.
 */
public void removeInterceptorFromAllInterfaces(String interceptorID) throws IllegalLifeCycleException,
        NoSuchInterfaceException, NoSuchComponentException, IllegalInterceptorException;
```

Most of these methods take as parameter, among others, an interceptorID. This identifier may be:

- The controller interface name of a controller that implements the Interceptor interface. For instance: myinterceptor-controller
- Or the concatenation of the name of a non functional component and the name of one of its server interfaces, that implements the Interceptor interface, separated by a point. For instance: InterceptorNFComponent.interceptor-services

Interceptors can also be attached to interfaces throught ADL. To do so, an additional attribute must be used in the interface tag of the interface on which to attach the interceptor(s). This attribute is interceptors and takes as value the list of interceptor identifiers that must act as interceptors for this interface. If the interceptor identifier represents a controller interface, the identifier must be prefixed by the "this." keyword like for bindings. If the interceptor identifier represents a server interface of a non functional component, the identifier is the same than the one used for the interceptor controller API (ie. the concatenation of the name of the non functional component and the name of its server interface). Each interceptor identifiers must be separated by a comma. When using this attribute, it becomes useless to explicitely add the interceptor controller to the list of controllers because it will be automatically added if it is not present by default.

Interceptors can be composed in a sequentially manner.

The beforeMethodInvocation method is called sequentially for each controller in the order they have been set with the interceptor controller or through ADL. The afterMethodInvocation method is called sequentially for each controller in the reverse order they have been set with the interceptor controller or through ADL.

For instance, if 2 interceptors are attached to an interface in this order:

```
Interceptor1 and then Interceptor2
```

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

```
—> caller —> Interceptor1.beforeMethodInvocation —> Interceptor2.beforeMethodInvocation —> callee.invocation —>
 Interceptor2.afterMethodInvocation —> Interceptor1.afterMethodInvocation
```

4.5.7.2. Writing a custom interceptor

To implement an interceptor as a controller object, it has to follow the rules explained in Section 4.5.1, “Controllers” for the creation of a custom controller. In addition of that, the controller object has to implement the Interceptor interface, which declares interception methods (pre/post interception) that have to be implemented.

Here is a simple example of a controller object acting as an interceptor:

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

import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.factory.InstantiationException;
import org.objectweb.fractal.api.type.TypeFactory;
import org.objectweb.proactive.core.ProActiveRuntimeException;
import org.objectweb.proactive.core.component.control.AbstractPAController;
import org.objectweb.proactive.core.component.interception.InterceptedRequest;
import org.objectweb.proactive.core.component.interception.Interceptor;
import org.objectweb.proactive.core.component.type.PAGCMTypeFactoryImpl;


public class MyInterceptor extends AbstractPAController implements Interceptor, ControllerItf {

   public MyInterceptor(Component owner) {
      super(owner);
   }

   protected void setControllerItfType() {
      try {
         setItfType(PAGCMTypeFactoryImpl.instance().createFcItfType("myinterceptor-controller",
               ControllerItf.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 InterceptedRequest beforeMethodInvocation(InterceptedRequest interceptedRequest) {
      System.out.println("pre processing an intercepted a functional invocation");
      // interception code

      return interceptedRequest;
   }

   public InterceptedRequest afterMethodInvocation(InterceptedRequest interceptedRequest) {
      System.out.println("post processing an intercepted a functional invocation");
      // interception code

      return interceptedRequest;
   }
}
```

Then for instance such an interceptor could be attached to an interface named "itf" by simply calling the addInterceptorOnInterface method of the interceptor controller:

```
Utils.getPAInterceptorController(component).addInterceptorOnInterface("itf", "myinterceptor-controller");
```

By using ADL, the interceptor could be attached to the interface like that:

```
<interface signature="signature-itf" role="server" name="itf" interceptors="this.myinterceptor-controller"/>
```

To implement an interceptor as a non functional component, it has to follow the rules in explained in Section 4.5.5, “Structuring the membrane with non-functional components” for the creation of a non functional component. In addition of that, the non functional component has to provide a functional server interface that implements the Interceptor interface, which declares interception methods (pre/post interception) that have to be implemented.

Here is a simple example of the ADL definition of a non functional component acting as an interceptor:

```
<definition name="org.objectweb.proactive.examples.components.interception.Interceptor">

 <interface signature="org.objectweb.proactive.core.component.interception.Interceptor" role="server" name="interceptor-
services"/>

   <content class="org.objectweb.proactive.examples.components.interception.InterceptorImpl"/>
</definition>
```

Assuming this non functional component is named InterceptorNFComponent, then for instance such an interceptor could be attached to an interface named "itf" by simply calling the addInterceptorOnInterface method of the interceptor controller:

```
Utils.getPAInterceptorController(component).addInterceptorOnInterface("itf", "InterceptorNFComponent.interceptor-services");
```

By using ADL, the interceptor could be attached to the interface like that:

```
<interface signature="signature-
itf" role="server" name="itf" interceptors="InterceptorNFComponent.interceptor-services"/>
```

Chapter 5. Architecture and design

The implementation of the Fractal/GCM model is achieved by reusing the extensible architecture of ProActive, notably the meta-object protocol and the management of the request queue. 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.

5.1. 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, whether active objects are local or remote.

The following diagram explains this mechanism:

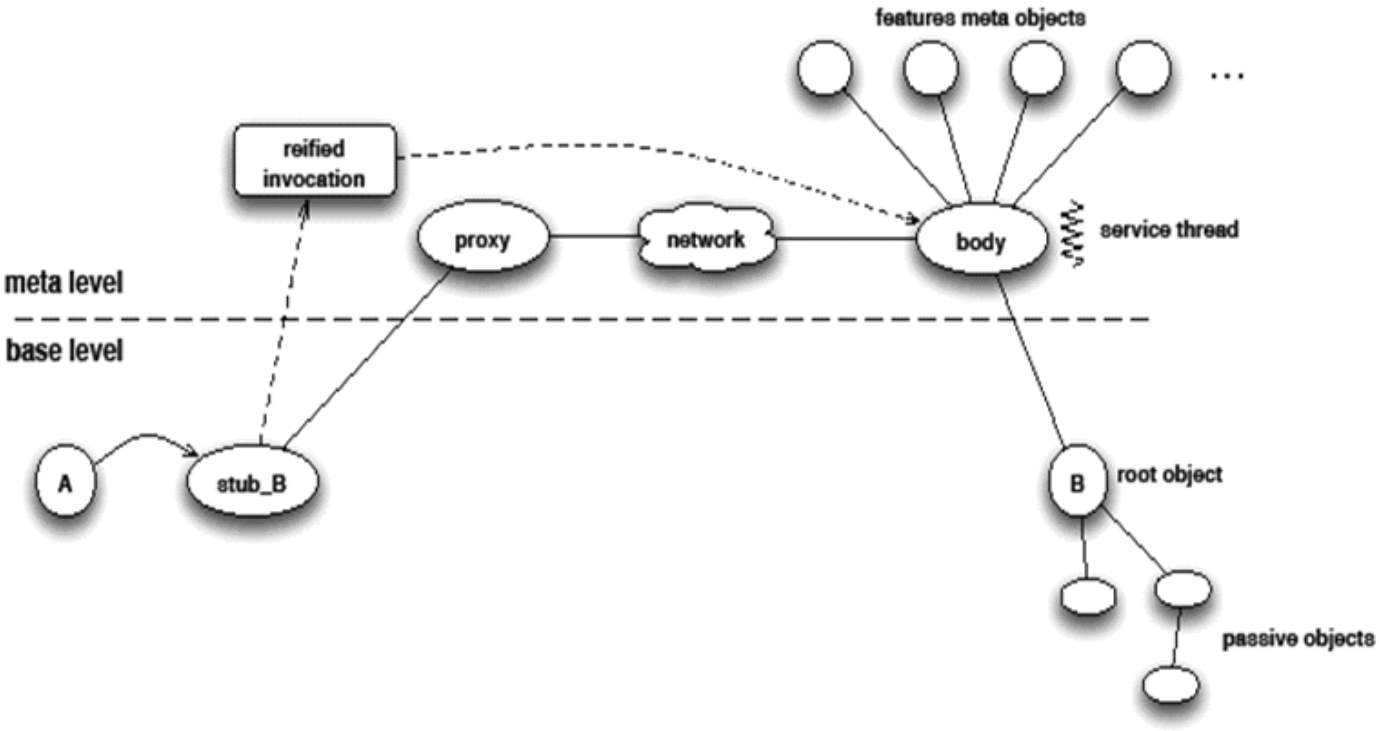


Figure 5.1. ProActive's Meta-Objects Protocol.

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 'a' has a reference on active object 'b' (of type B) through a stub (of type Stub_B) because it is generated as a subclass of B and a proxy. When 'a' invokes a method on 'stub_B', 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 'b'.

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:

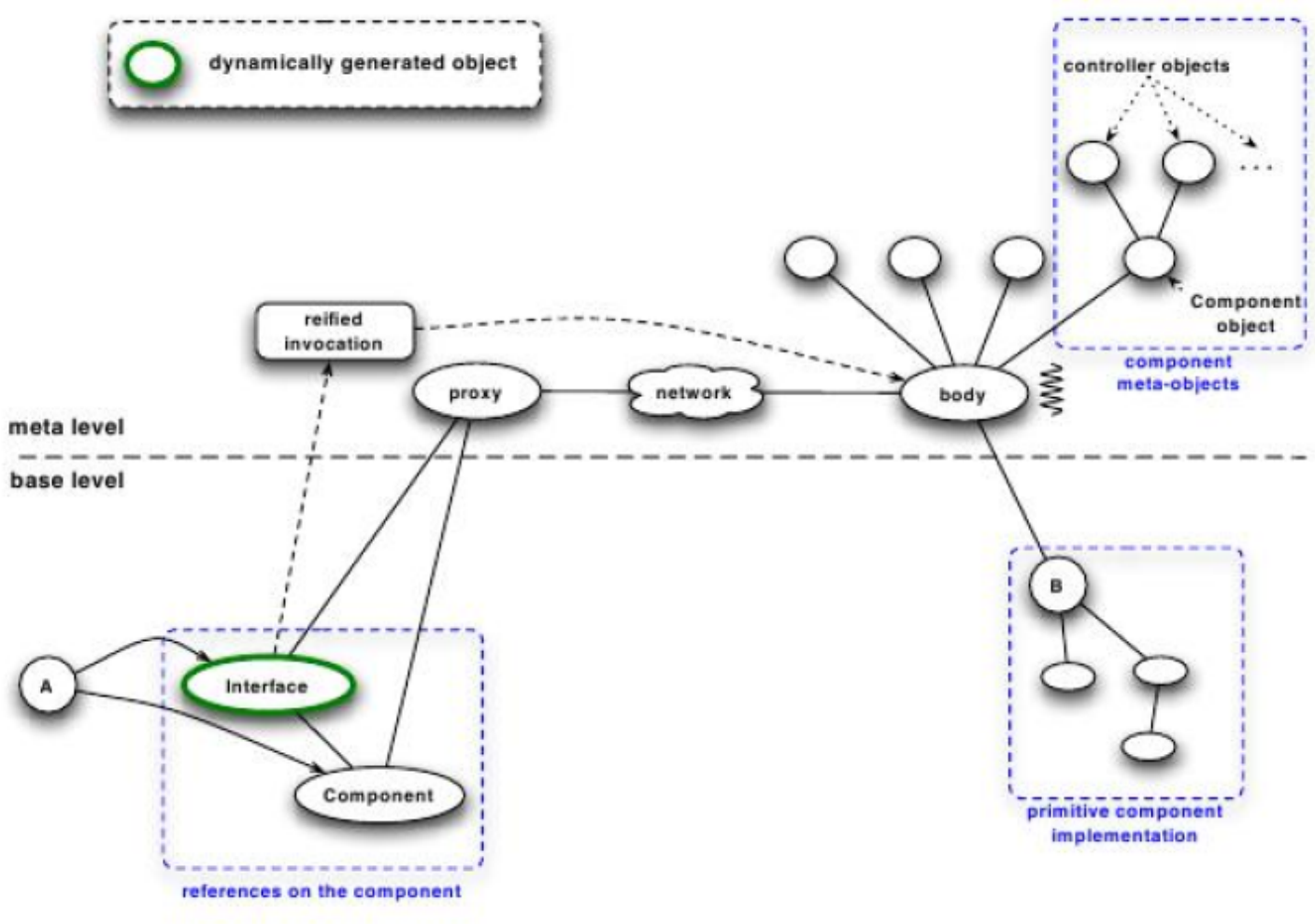


Figure 5.2. The ProActive MOP with component meta-objects and component representative

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 'b'. The standard ProActive stub (that gives a representation of type B on the figure) is not used here, as we manipulate components. In Fractal/GCM, a reference on a component is of type Component, and references to interfaces are of type Interface. 'a' can now manipulate the component based on 'b' 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/GCM interfaces are reified and transmitted (possibly through a network) to the body of the active object corresponding to the involved component. All standard operations of the Fractal/GCM API are now accessible.

5.2. 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), whether they are 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. This class is an empty class, because for composite components, all the action takes place in the meta-level. But it is used as a base to build active objects, and its name helps to identify it with the IC2D visual monitoring tool.

5.3. Method invocations on component interfaces

Invoking a method on an active object means invoking a method on the stub of this active object. What usually happens 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.

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

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

This lifecycle is implemented by customizing the activity of the active objects. In the context of components, we distinguish the component activity (the non-functional activity) from the functional activity. The component activity corresponds to the stopped state of the lifecycle of the component (i.e. only control requests are served). The functional activity is encapsulated and starts when the lifecycle is started. The default behaviour is to serve all control requests in a FIFO order until the component is started using the lifecycle controller. Then, a component serves all requests, control and functional, in a FIFO order, until the lifecycle is stopped. The functional activity is encapsulated in the component activity. This is illustrated in Figure 5.3, “Default component activity”.

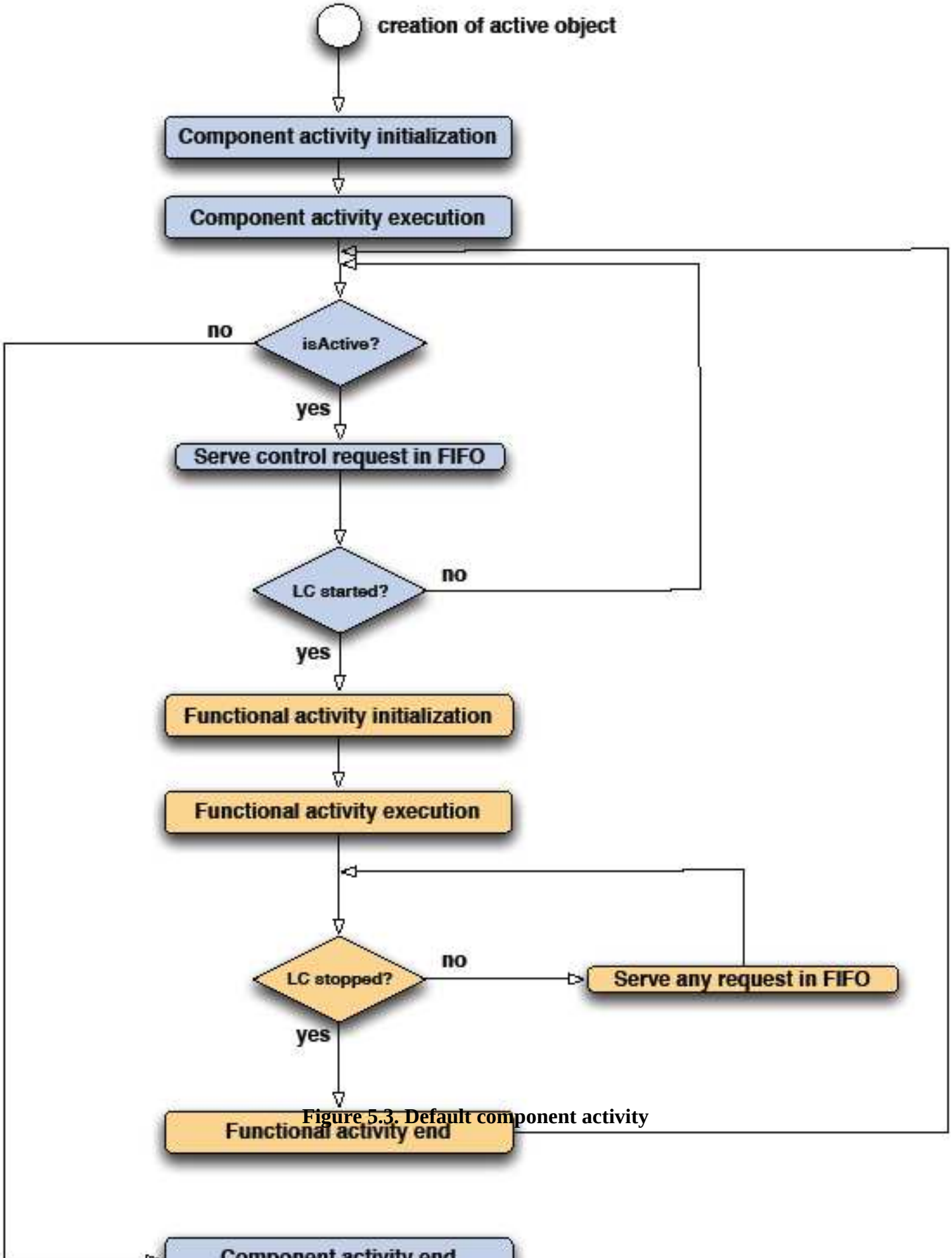


Figure 5.3. Default component activity

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

ProActive offers the possibility to customize the activity of an active object; this is actually a fundamental feature of the library, as it allows to fully specify the behaviour of active objects. Thus, in term of components, the component activity may be customized by implementing the ComponentInitActive, ComponentRunActive and ComponentEndActive java interfaces. By default, the component activity initialization and the component activity termination are done only one time:

- The initialization phase is done during the instantiation of the component (directly followed by the component activity execution). From this moment on, the component is in the active state in term of activity of active object (the isActive() condition on Figure 5.3, “Default component activity” is true).
- The termination phase is done when the isActive() condition is false, i.e. when the terminate method of the active object representing the component is called.

Second, the functional activity may also be customized by implementing the InitActive, RunActive and EndActive interfaces. Two conditions must be respected though, for a smooth integration with the component lifecycle:

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

By default, when the lifecycle is started, the functional activity is initialized, run, then ended when the isActive() condition is false, i.e. when the lifecycle is stopped.

Chapter 6. Component examples

Two examples are presented in this chapter: code snippets for visualizing the transition between active objects and components, and the 'hello world' from the Fractal tutorial. The programming model is Fractal/GCM, and one should refer to the Fractal documentation for more detailed examples.

6.1. 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 bar(); // dummy method

}
```

'AImpl' is the class implementing this interface:

```
public class AImpl implements A {

   Service service;

   public AImpl() {
   }

   public String bar() {
      return "bar";
   }


}
```

The class is then instantiated in a standard way:

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

Active objects are instantiated using factory methods from the PAActiveObject class (see Chapter 6.1. Simple Computation And Monitoring Agent[1]). 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 activeObject = (A) PAActiveObject.newActive(AImpl.class.getName(), // 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 controllers.

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../GetStarted/multiple_html/ActiveObjectTutorial.html#SimpleCMA

6.1.1. Type

The type of a 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 application components. In order to do this, we have first to 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 gcm.provider system property or in the fractal.provider system property if the first one has not been set, and uses this instance to get the bootstrap component):

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

Then, we get the GCMTypeFactory interface provided by this bootstrap component:

```
GCMTypeFactory tf = GCM.getGCMTypeFactory(boot);
```

Next, we can 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) });
```

6.1.2. 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.class.getName(), // signature of the base class
    new Object[] {} // Object[]
);
```

6.1.3. Description of the controllers

Properties relative to the controllers 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 or COMPOSITE
);
```

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

```
PAGenericFactory componentFactory = Utils.getPAGenericFactory(boot);
Component component = componentFactory.newFcInstance(aType, // type of the component (defining the client and
 server interfaces)
        controllerDesc, // implementation-specific description for the controller
        contentDesc, // implementation-specific description for the content
        aNode // location, could also be a virtual node
        );
```

6.1.4. 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 one:

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

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

And the AImpl class is:

```
public class AImpl implements A {

   Service service;

   public AImpl() {
   }


   // implementation of the A interface
   public void foo() {
      service.print("Hello World"); // 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;
        }
    }
}
```

6.2. 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.

6.2.1. Set-up

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

The code is almost identical to the Fractal tutorial's example[2].

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/GCM 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).
- Components can be distributed
- the ClientImpl provides an empty no-args constructor.

6.2.2. 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).

[2] http://fractal.objectweb.org/tutorials/fractal/index.html

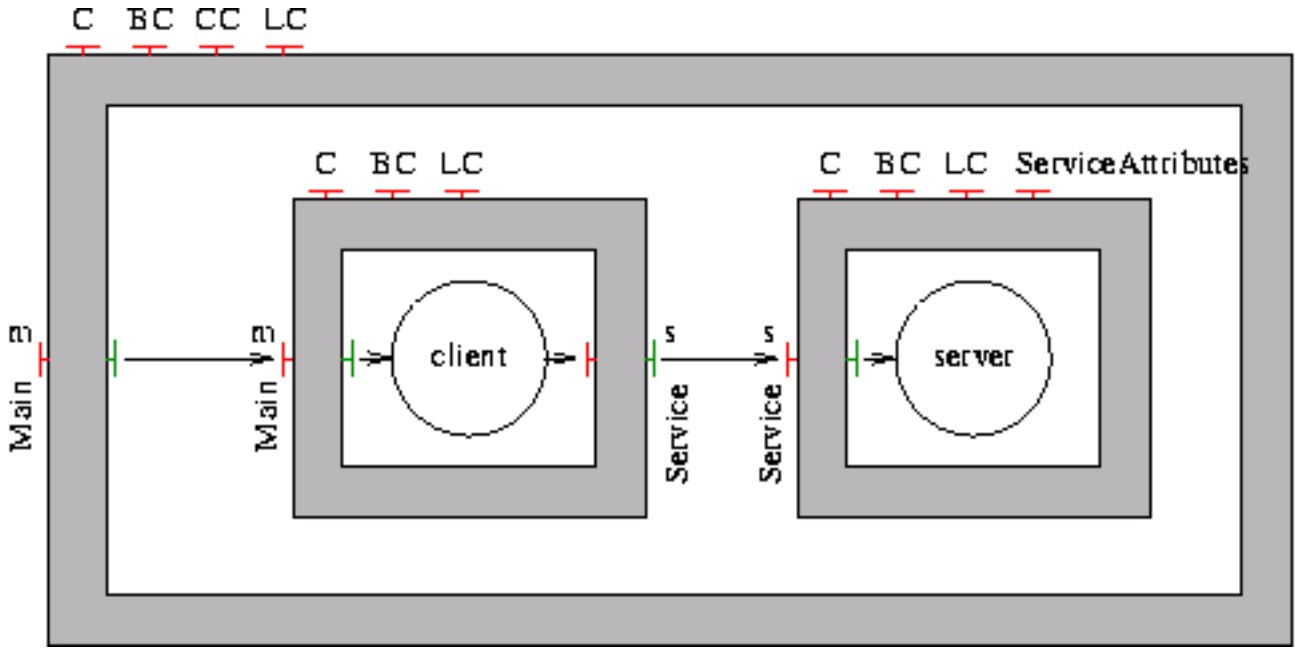


Figure 6.1. Client and Server wrapped in composite components (C and S)

6.2.3. Distributed deployment

This section is specific to the ProActive/GCM 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 Chapter 15. XML Deployment Descriptors[3]).

Note

This example has been written using the old deployment descriptors. However, it is obviously possible (and recommended) to write it using the new GCM Deployment descriptors. Besides, this examples will probably be updated with the new deployment version in the future.

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:

[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/Components/pdf/../../ReferenceManual/multiple_html/XML_Descriptors.html#XML_Descriptors

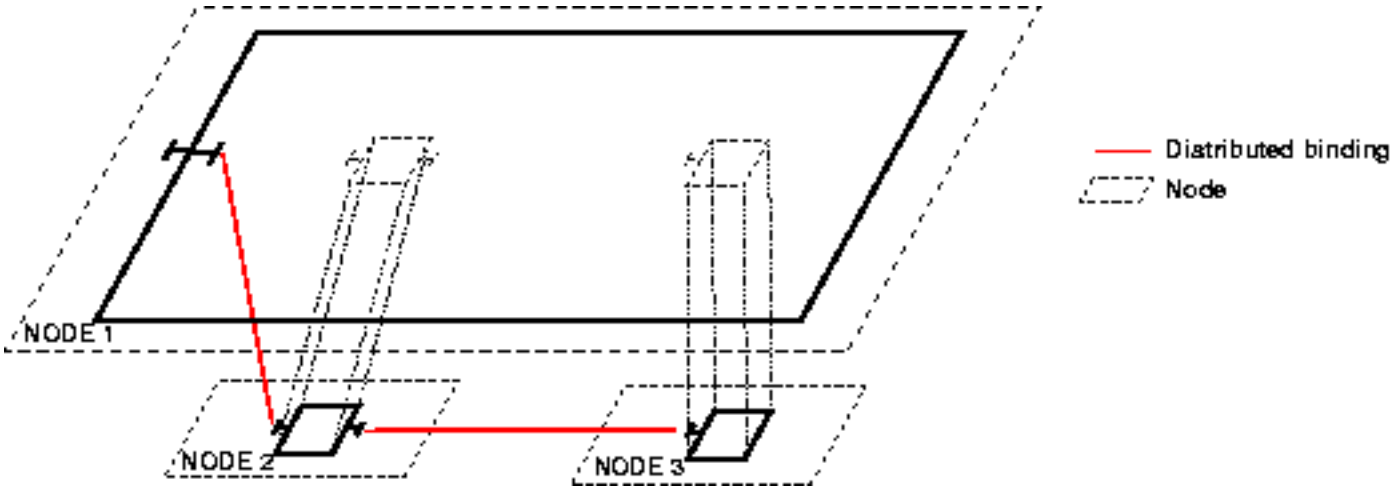


Figure 6.2. Without wrappers, the primitive components are distributed

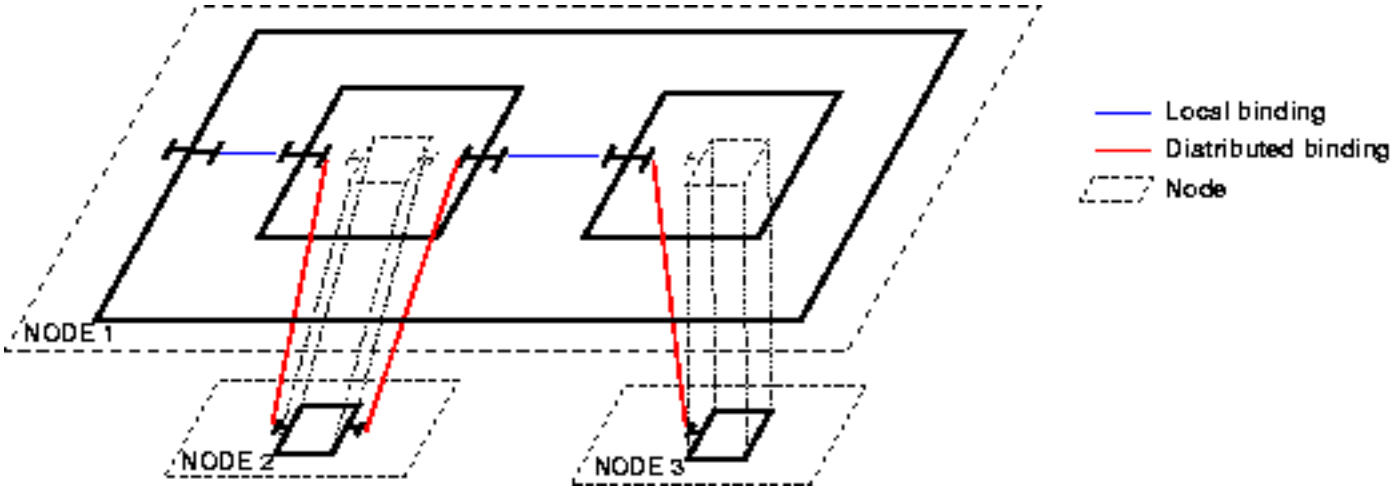


Figure 6.3. With wrappers only the primitive components are distributed

By default, 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. However, optimizations 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. For further information about optimizations, please refer to Section 4.5.6, “Short cuts”.

6.2.4. 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 gcm.provider system property (or 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:

```
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.mop] Generating class :
 8:   pa.stub.org.objectweb.proactive.core.component.type._StubComposite
 9: [INFO proactive.mop] Generating class :
10:   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
```

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
```

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 Registry on port
6646
32: [INFO proactive.mop] Generating class :
33:   pa.stub.org.objectweb.proactive.core.component.type._StubComposite
```

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 'saturn.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 as for a local deployment, the activity of active objects is independent from its location.

```
1: [INFO deployment.log] Server: print method called
```

```
 2: [INFO deployment.log] at
 3:   org.objectweb.proactive.examples.components.helloworld.ServerImpl.print(ServerImpl.java:45)
 4: [INFO deployment.log] at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 5: [INFO deployment.log] at
 6:   sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 7: [INFO deployment.log] at
 8:   sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
 9: [INFO deployment.log] at java.lang.reflect.Method.invoke(Method.java:597)
10: [INFO deployment.log] at org.objectweb.proactive.core.mop.MethodCall.execute(MethodCall.java:390)
11: [INFO deployment.log] at
12:   org.objectweb.proactive.core.component.request.ComponentRequestImpl.
13:     serveInternal(ComponentRequestImpl.java:176)
14: [INFO deployment.log] at
15:   org.objectweb.proactive.core.body.request.RequestImpl.serve(RequestImpl.java:170)
16: [INFO deployment.log] at
17:   org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.
18:     serveInternal(BodyImpl.java:539)
19: [INFO deployment.log] at
20:   org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.serve(BodyImpl.java:510)
21: [INFO deployment.log] at
22:   org.objectweb.proactive.core.body.AbstractBody.serve(AbstractBody.java:909)
23: [INFO deployment.log] at org.objectweb.proactive.Service.blockingServeOldest(Service.java:175)
24: [INFO deployment.log] at org.objectweb.proactive.Service.blockingServeOldest(Service.java:150)
25: [INFO deployment.log] at org.objectweb.proactive.Service.fifoServing(Service.java:126)
26: [INFO deployment.log] at
27:   org.objectweb.proactive.core.component.body.ComponentActivity$ComponentFIFORunActive.
28:     runActivity(ComponentActivity.java:226)
29: [INFO deployment.log] at
30:   org.objectweb.proactive.core.component.body.ComponentActivity.
31:     runActivity(ComponentActivity.java:183)
32: [INFO deployment.log] at
33:   org.objectweb.proactive.core.component.body.ComponentActivity.
34:     runActivity(ComponentActivity.java:183)
35: [INFO deployment.log] at org.objectweb.proactive.core.body.ActiveBody.run(ActiveBody.java:192)
36: [INFO deployment.log] at java.lang.Thread.run(Thread.java:619)
37: [INFO deployment.log] Server: begin printing...
38: [INFO deployment.log] ->hello world
39: [INFO deployment.log] Server: print done.
40:------------------------------------------------------------
```

6.2.5. The HelloWorld ADL files

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

```
<?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"/>
  <!-- @snippet-start exported_virtual_node_1 -->
  <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>
      <!-- @snippet-end exported_virtual_node_1 -->
    <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:

```
<?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.proac
    <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:

```
<?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.proact
    <!-- @snippet-start exported_virtual_node_2 -->
    <exportedVirtualNodes>
      <exportedVirtualNode name="client-node">
        <composedFrom>
          <composingVirtualNode component="this" name="client-node"/>
```

```
        </composedFrom>
      </exportedVirtualNode>
    </exportedVirtualNodes>
    <!-- @snippet-break exported_virtual_node_2 -->
    <content class="org.objectweb.proactive.examples.components.helloworld.ClientImpl"/>
    <!-- @snippet-resume exported_virtual_node_2 -->
    <virtual-node name="client-node"/>
    <!-- @snippet-end exported_virtual_node_2 -->
</definition>
```

org.objectweb.proactive.examples.components.helloworld.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.proac
    <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"/>
</definition>
```

Chapter 7. Component perspectives: a support for advanced research

The ProActive/GCM framework is a functional and flexible implementation of the Fractal/GCM API and model. One can configure and deploy a system of distributed components, including Grids.

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

7.1. 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/GCM ones, and more autonomy. By providing higher level of primitives, the principal aim is to help the programmer to design safe scenarios. 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/GCM. 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 information necessary to fulfill the other one). Then, between those two requests, no reconfiguration can occur if it involves the binding used for the requests.

As a consequence, it is important to design 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/GCM.

The autonomic computing paradigm is related to this challenge because it 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. Autonomy of components represents a key asset for large scale distributed computing.

7.2. 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. Automatons can be generated from behavioral information and structural description.
3. Model checking tools are used to verify the automaton.

The Vercors [1] platform investigates such kinds of scenarios.

[1] http://www-sop.inria.fr/oasis/Vercors/

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 (Plasil02), Fractal (Fractal04) and CORBA Component Model (CCM) 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 JavaA05, InterfaceAutomata2001, Reussnerm, FACS-06 or that describe behaviour of components JKP05.

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 similar to ifip05, STSLib07. 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).

7.3. Pattern-based deployment

Distributed computational applications are designed by defining a functional (or domain) decomposition, and this decomposition often presents 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/GCM ADL, particularly for systems that contain parameterized numbers of identical components.

7.4. Graphical tools

We are developing the VCE (Vercors Component Environment), 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. Fractal/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 Proactive/GCM, in particular for expressing request queue selection, and multicast/gathercast policies.

Part III. Back matters

Table of Contents

Chapter 8. Appendix

8.1. The GCM Basics example files

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

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
package org.objectweb.proactive.examples.components.userguide.starter;

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

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

```
/*
 * ################################################################
 *
```

```
* ProActive Parallel Suite(TM): The Java(TM) library for
*    Parallel, Distributed, Multi-Core Computing for
*    Enterprise Grids & Clouds
*
* Copyright (C) 1997-2012 INRIA/University of
*                 Nice-Sophia Antipolis/ActiveEon
* Contact: proactive@ow2.org or contact@activeeon.com
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Affero General Public License
* as published by the Free Software Foundation; version 3 of
* the License.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* Affero General Public License for more details.
*
* You should have received a copy of the GNU Affero General Public License
* along with this library; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
* USA
*
* If needed, contact us to obtain a release under GPL Version 2 or 3
* or a different license than the AGPL.
*
*  Initial developer(s):               The ProActive Team
*                        http://proactive.inria.fr/team_members.htm
*  Contributor(s):
*
* ################################################################
* $$PROACTIVE_INITIAL_DEV$$
*/
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"/>
</definition>
```

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

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
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"/>
</definition>
```

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

```
/*
 * ################################################################
```

```
*
* ProActive Parallel Suite(TM): The Java(TM) library for
*    Parallel, Distributed, Multi-Core Computing for
*    Enterprise Grids & Clouds
*
* Copyright (C) 1997-2012 INRIA/University of
*                 Nice-Sophia Antipolis/ActiveEon
* Contact: proactive@ow2.org or contact@activeeon.com
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Affero General Public License
* as published by the Free Software Foundation; version 3 of
* the License.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* Affero General Public License for more details.
*
* You should have received a copy of the GNU Affero General Public License
* along with this library; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
* USA
*
* If needed, contact us to obtain a release under GPL Version 2 or 3
* or a different license than the AGPL.
*
*  Initial developer(s):               The ProActive Team
*                        http://proactive.inria.fr/team_members.htm
*  Contributor(s):
*
* ################################################################
* $$PROACTIVE_INITIAL_DEV$$
*/
package org.objectweb.proactive.examples.components.userguide.starter;

import java.util.HashMap;

import org.etsi.uri.gcm.util.GCM;
import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.api.control.BindingController;
import org.objectweb.proactive.core.component.adl.FactoryFactory;
import org.objectweb.proactive.core.component.adl.Registry;
import org.objectweb.proactive.core.config.CentralPAPropertyRepository;
import org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment;
import org.objectweb.proactive.gcmdeployment.GCMApplication;


public class Main {
   public static void main(String[] args) throws Exception {
      CentralPAPropertyRepository.GCM_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 = GCM.getBindingController(client);
        bc.bindFc("s", server.getFcInterface("s"));

        // start components
        GCM.getGCMLifeCycleController(server).startFc();
        GCM.getGCMLifeCycleController(client).startFc();

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

        // stop components
        GCM.getGCMLifeCycleController(client).stopFc();
        GCM.getGCMLifeCycleController(server).stopFc();

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

org.objectweb.proactive.examples.components.userguide.starter.deploymentDescriptor.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.home" />
   </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>
```

Bibliography

[ACC05] Isabelle Attali, Denis Caromel, and Arnaud Contes. *Deployment-based security for grid applications*. The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25. . LNCS. 2005. Springer Verlag.

[BBC02] Laurent Baduel, Francoise Baude, and Denis Caromel. *Efficient, Flexible, and Typed Group Communications in Java*. 28--36. Joint ACM Java Grande - ISCOPE 2002 Conference. Seattle. . 2002. ACM Press. ISBN 1-58113-559-8.

[BBC05] Laurent Baduel, Francoise Baude, and Denis Caromel. *Object-Oriented SPMD*. Proceedings of Cluster Computing and Grid. Cardiff, United Kingdom. . May 2005.

[BCDH05] Francoise Baude, Denis Caromel, Christian Delbe, and Ludovic Henrio. *A hybrid message logging-cic protocol for constrained checkpointability*. 644--653. Proceedings of EuroPar2005. Lisbon, Portugal. . LNCS. August-September 2005. Springer Verlag.

[BCHV00] Francoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. *Communicating mobile active objects in java*. 633--643. http://www-sop.inria.fr/oasis/Julien.Vayssiere/publications/18230633.pdf. Proceedings of HPCN Europe 2000. . LNCS 1823. May 2000. Springer Verlag.

[BCM+02] Francoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssiere. *Interactive and descriptor-based deployment of object-oriented grid applications*. 93--102. http://www-sop.inria.fr/oasis/personnel/Julien.Vayssiere/publications/hpdc2002_vayssiere.pdf. Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. Edinburgh, Scotland. . July 2002. IEEE Computer Society.

[BCM03] Francoise Baude, Denis Caromel, and Matthieu Morel. *From distributed objects to hierarchical grid components*. http://proactive.activeeon.com//userfiles/file/papers/HierarchicalGridComponents.pdf. International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November. Springer Verlag. . 2003. Lecture Notes in Computer Science, LNCS. ISBN ??.

[Car93] Denis Caromel. *Toward a method of object-oriented concurrent programming*. 90--102. http://citeseer.ist.psu.edu/300829.html. *Communications of the ACM*. 36. 9. 1993.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer Verlag. 2005.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Serpette. *Asynchronous and deterministic objects*. 123--134. http://doi.acm.org/10.1145/964001.964012. Proceedings of the 31st ACM Symposium on Principles of Programming Languages. . 2004. ACM Press.

[CKV98a] Denis Caromel, W. Klauser, and Julien Vayssiere. *Towards seamless computing and metacomputing in java*. 1043--1061. http://proactive.inria.fr/doc/javallCPE.ps. Concurrency Practice and Experience. . Geoffrey C. Fox. 10, (11--13). September-November 1998. Wiley and Sons, Ltd..

[HCB04] Fabrice Huet, Denis Caromel, and Henri E. Bal. *A High Performance Java Middleware with a Real Application*. http://proactive.inria.fr/doc/sc2004.pdf. Proceedings of the Supercomputing conference. Pittsburgh, Pensylvania, USA. . November 2004.

[BCDH04] F. Baude, D. Caromel, C. Delbe, and L. Henrio. *A fault tolerance protocol for asp calculus : Design and proof*. http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5246.pdf. Technical ReportRR-5246. INRIA. 2004.

[FKTT98] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. *A security architecture for computational grids*. 83--92. http://citeseer.ist.psu.edu/foster98security.html. ACM Conference on Computer and Communications Security. . 1998.

[CDD06c] Denis Caromel, Christian Delbe, and Alexandre di Costanzo. *Peer-to-Peer and Fault-Tolerance: Towards Deployment Based Technical Services* . Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture . Paris, France. . January 2006.

[CCDMCompFrame06] Denis Caromel, Alexandre di Costanzo, Christian Delbe, and Matthieu Morel. *Dynamically-Fulfilled Application Constraints through Technical Services - Towards Flexible Component Deployments* . Proceedings of HPC-GECO/CompFrame 2006, HPC Grid programming Environments and COmponents - Component and Framework Technology in High-Performance and Scientific Computing . Paris, France. . June 2006. IEEE.

[CCMPARCO07] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. *Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines*. *Parallel Computing Journal on Large Scale Grid*. 2007.

[PhD-Morel] Matthieu Morel. *Components for Grid Computing* . http://www-sop.inria.fr/oasis/personnel/Matthieu.Morel/publis/phd_thesis_matthieu_morel.pdf. PhD thesis. University of Nice Sophia-Antipolis. 2006.

[FACS-06] I. \vCern\'a, P. Va\vrekov\'a, and B. Zimmerova. "Component Substitutability via Equivalencies of Component-Interaction Automata". To appear in ENTCS. 2006.

[JavaA05] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. "A Component Model for Architectural Programming". 2005.

[Fractal04] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. "An Open Component Model and Its Support in Java.". 2004.

[ifip05] Alessandro Coglio and Cordell Green. "A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Appplets". 2005.

[STSLib07] Fabricio Fernandes and Jean-Claude Royer. "The STSLIB Project: Towards a Formal Component Model Based on STS". To appear in ENTCS. 2007.

[InterfaceAutomata2001] Luca de Alfaro, Tom Henzinger. "Interface automata". 2001.

[CCM] OMG. "CORBA components, version 3". 2002.

[Plasil02] F. Plasil and S. Visnovsky. "Behavior Protocols for Software Components". *IEEE Transactions on Software Engineering*. 28. 2002.

[Reussnerm] Reussner, Ralf H.. "Enhanced Component Interfaces to Support Dynamic Adaption and Extension". IEEE. 2001.

[JKP05] P. Jezek, J. Kofron, F. Plasil. "Model Checking of Component Behavior Specification: A Real Life Experience". Electronic Notes in Theoretical Computer Science (ENTCS). 2005.

[MB01] V. Mencl and T. Bures. "Microcomponent-based component controllers: A foundation for component aspects". APSEC. . Dec. 2005. IEEE Computer Society.

[SPC01] L. Seinturier, N. Pessemier, and T. Coupaye. "AOKell: an Aspect-Oriented Implementation of the Fractal Specifications". 2005.

Index