Project no.FP6-034442

GridCOMP

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

STREP Project

Advanced Grid Technologies, Systems and Services

D.NFCF.02 – NFCF early prototype (with basic non functional features)

Due date of deliverable: May 31^{th}, 2007

Actual submission date: June 26^{th} 2007

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

Organisation name of lead contractor for this deliverable: UNIPI

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PP	Restricted	PP

Keyword list: autonomic management, component controller, GMC, task farm
Responsible Partner: UNIPI

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	25-06-2007	Draft	Marco ALDINUCCI
1	04-07-2007	Draft	Marco ALDINUCCI
2	11-07-2007	Draft	Marco ALDINUCCI

Deliverable manager

- Marco Aldinucci, UNIPI

List of Contributors

- Sonia Campa, UNIPI
- Patrizio Dazzi, ISTI-CNR

List of Evaluators

- Françoise Baude, INRIA
- Rajkumar Buyya, U. MELBOURNE

Executive Summary

The D.NFCF.01 deliverable released in the context of the GridCOMP project [1], provides an architectural specification of the non-functional component subsystem of GCM and introduce the behavioural skeleton framework as an abstraction for leading the development of GCM applications. A behavioural skeleton exploits two levels of adaptivity: *i*) a passive level, in which autonomic operations are provided as a set of limited and well-defined primitives; *ii*) an active level, in which the adaptive behavior of the component is leaded by a manager who takes care of planning and taking adaptive decisions. A user can write her autonomic application on top of both layers, depending on the level of autonomicity he needs.

While the D.NFCF.01 focuses on the general structure of behavioural skeletons, this deliverable aims at presenting a first prototype of such autonomic subsystem on top of which we have developed some applicative examples for structuring a parallel application through the proposed framework.

In particular, the deliverable focuses on different implementations of the farm skeleton, a well-known pattern of parallelism we have implemented in a variety of use cases within the prototype, as a readily usable adaptive component, by exploiting both ProActive collective interfaces and single server interfaces.

The architectural aspects of the prototype has been give in in the D.NFCF.01 deliverable to which this tutorial is consequently related. The code sources have been distributed as a zip archive with the current document.

Contents

1 Introduction

The objective of this tutorial is to illustrate how to define, to describe, to compile and to run an autonomic application in which autonomic components and their related autonomic controllers can be easily configured.

The tutorial offers four programming experiences distributed as a zipped archive file within this document. The examples have been developed on top of the behavioural skeleton subsystem we provide in a prototype version. All the sample applications focus on the implementation of an application using a task farm for distributing working tasks. At first, the farm skeleton component has been defined by implementing a multicast interface natively provided by ProActive for distributing the tasks to a set of workers (sub-components). Since both the management policies and the implementation related to these interfaces has revealed to be inadequate to fully express the behavioural skeleton specification (particularly, they do not support stream/event communication), we have provided a subsystem in which a basic implementation of the farm component can be used, extended and/or customized by the user. The use cases here presented can be summarized as follows:

- an autonomic application in which the component farm is implemented by using the collective server interface `Multicast` natively offered by ProActive; in this version, a stream of input tasks is emulated through arguments given to the multicast interface and the multicast interface methods do not return values in order to fully emulate a stream communication (package `multicusecase` in folder `MulticastProj`);
- an autonomic application in which the component farm is implemented by using the collective server interface `Multicast` that exposes synchronous invocations, i.e. its methods return values (package `multicusecaseSynch` in folder `MulticastProj`);
- an autonomic application in which autonomicity is exploited at a passive level, allowing the interaction with an autonomic controller to which the user can ask for increasing/decreasing the parallelism degree by manually pushing commands from the shell (package `passivefarm` in folder `Passive`);
- an autonomic application in which autonomicity is exploited at an active level. A component manager takes care about monitoring the overall computation time and it increases or decreases the parallelism degree (interacting with the autonomic controller) in order to keep the statically given QoS requirements (package `activefarm` in folder `Active`).

`MulticastProj`, `Passive` and `Active` represent an Eclipse Project each and they all use as required project on the build path the `OrgImport` folder content. In other words, `OrgImport` contains the "core" classes and files for the development of GCM autonomic component applications, while the other projects represent the use case we will detail in this tutorial.

2 Implementing an asynchronous farm using the Multicast interface

The main goal of the experiments included in the package `multicusecase` is to implement a farm in which a stream parallel computation is emulated by means of a method call to the native multicast interface provided by ProActive. In order to fully describe a stream communication, each method call to the multicast interface returns a `void` type.

2.1 Structure of the package `multicusecase`

This package is composed by 4 sub-packages, as detailed in Tab. 1

Name of the sub-package	Description of the sub-package
`adl`	contains the ADL descriptors configuring the application (with .fractal extension)
`descriptor`	contains the XML files and the implementation files implementing the AutonomicController
`impl`	contains the user java implementation classes
`itf`	contains the user java interface classes

Table 1: Composition of the `multicusecase` package

2.2 Description of the application

The application is depicted in Fig.1

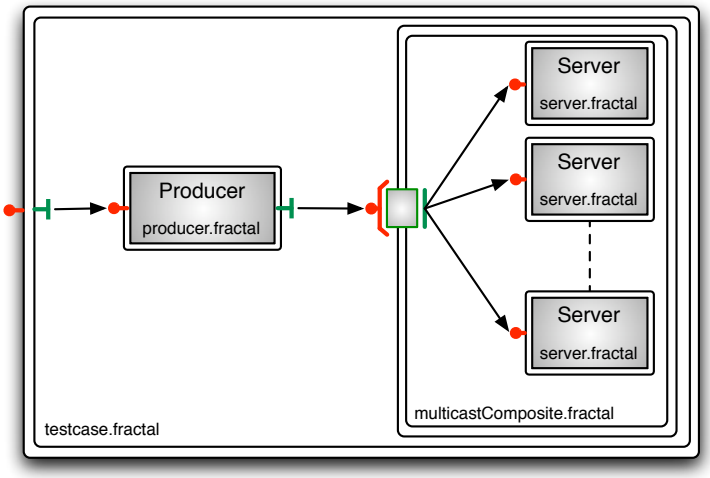

Figure 1: Asynchronous farm application

The application component (described in the `testcase.fractal` file) exposes a server interface offering a service: each invocation to such server interface activates a `Producer` (described by the `producer.fractal` file) component that is in charge of managing the stream of requests to be sent to the farm. The stream is emulated by means of lists of tasks. The farm is represented by the composite component described in the `multicastComposite.fractal` file; it exposes a server interface offering a (parallel) service through a multicast interface and includes a number of sub-components implementing the workers of the farm. Each request sent to the composite component through a method call to the multicast interface provides a list of tasks as arguments: such tasks are scheduled among the available workers exploiting the round-robin policy natively offered by ProActive.

Notice that this structure is simpler than the one proposed in the D.NFCF.01 document because the goal of this application is to shape how multicast interfaces

work and how they can be used to describe our farm component. In the next section we will provide a more complex structure example.

Pragmatically, the ADL file `testcase.adl` included into the `adl` package is the file describing the application and it is structured as follows.

```
<?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="multicusecase.adl.testcase">

<interface signature="multicusecase.itf.Producer"
                   role="server"
                   name="runTestItf"/>

<!-- the producer component -->
<component name="tester"
                   definition="multicusecase.adl.producer"/>

<!-- second component: the farm -->
<component name="multicastComposite"
                   definition="multicusecase.adl.multicastComposite"/>

<!-- the internal bindings -->
        <binding client="this.runTestItf"
                 server="tester.runTestItf"/>
        <binding client="tester.clientItf"
                 server="multicastComposite.multicastServerItf"/>

        <!-- declare the type of the component -->
        <controller desc="composite"/>
</definition>
```

The first line declares the name of the current file in the context of the package.

```
<definition name="multicusecase.adl.testcase">
```

A composite component definition is represented by:

- the interface(s) the component offers. As mentioned above, our application offers just a server interface (the one that we will call in order to get the application running), whose name is `runTestItf` and the signature is defined in the java class `multicusecase.itf.Producer`. These information are detailed by the `interface` tag:

  ```
  <interface signature="multicusecase.itf.Producer"
             role="server"
             name="runTestItf"/>
  ```

- a set of references to the ADL files describing the inner components (one per each inner component). The entry describing each sub-component (the producer and the farm in our case) is given by the `component` tag and it specifies the logic name of the component and the path of the ADL file describing it. As an example,

  ```
  <component name="tester"
             definition="multicusecase.adl.producer"/>
  ```

defines a component whose logical name in the context of this file is `tester`, and whose ADL file descriptor is included in the `multicusecase.adl` package and is named `producer.adl`.

- the bindings between the current internal interfaces and the sub-components. In our case, there is just one internal binding between the internal client interface `runTestItf` and the producer's server interface `runTestItf` belonging to the logical sub-component named `tester`.

  ```
  <binding client="this.runTestItf"
           server="tester.runTestItf"/>}
  ```

  By looking backward at the definition of the sub-components, you will notice that `tester` is the producer sub-component and you will need to access the file `multicusecase.adl.producer` to appreciate the details of its server interface named `runTestItf`.

- the external bindings between the inner components. Again, in this piece of code you will have a sequel of `<binding ...>` tags in which a server and a client interface will be exploited, on the basis of the static binding that your application will set up at launching time.

  In our case, the client interface of the sub-component `tester` named `clientItf` will be linked to the server interface named `multicastServerItf` belonging to the `multicastComposite` subcomponent:

  ```
  <binding client="tester.clientItf"
           server="multicastComposite.multicastServerItf"/>}
  ```

  At the end, the type of the native component application's controller is declared as been `composite`.

The ADL description of the farm component The farm component is described in the `multicusecase.adl.multicastComposite` file.

```
<definition name="multicusecase.adl.multicastComposite">

  <!-- interface declarations ->
   <interface signature="multicusecase.itf.MulticastTestItf"
              role="server"
              name="multicastServerItf"
              cardinality="multicast"/>

  <!-- inner subcomponents ->
   <component name="server0" definition="multicusecase.adl.server(0)"/>
   <component name="server1" definition="multicusecase.adl.server(1)"/>
   <component name="server2" definition="multicusecase.adl.server(2)"/>
   <component name="server3" definition="multicusecase.adl.server(3)"/>
   <component name="server4" definition="multicusecase.adl.server(4)"/>

  <!-- internal bindings ->
   <binding client="this.multicastServerItf" server="server0.serverItf"/>
   <binding client="this.multicastServerItf" server="server1.serverItf"/>
   <binding client="this.multicastServerItf" server="server2.serverItf"/>
   <binding client="this.multicastServerItf" server="server3.serverItf"/>
   <binding client="this.multicastServerItf" server="server4.serverItf"/>

  <!-- reference to the xml file describing the Autonomic controller for
```

```
      this component->
  <controller desc="/multicusecase/descriptor/
                                  AutonomicControllerMulticastItf.xml"/>
</definition>
```

At this point, you should be able to understand the logic behind these definitions. We will just point out two entries:

- the `multicastComposite` component exports a *server multicast* interface only, and such type of interface must be declared. Thus, in the `<interface ...>` tag, it *must* be declared that the `cardinality` of the interface is `multicast` (instead of the default `singleton`).
- in the `controller` section entry we want to specify our-own set of controllers. The file `AutonomicControllerMulticastItf.xml` appearing in the `<controller ...>` entry is an XML file (detailed later) containing a set of references to the interfaces and the classes implementing the component's controllers.

All the other components included in the application are primitive components. We will detail the component description representing a *worker* of the farm but the same considerations apply also for the definition of the `Producer` component.

Each worker is implemented by a primitive component; the ADL description of such component is given in the `multicusecase.adl.server` file.

```
<?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="multicusecase.adl.server" arguments="id">

   <!-- the server interface offering a service ->
   <interface signature="multicusecase.itf.ServerTestItf"
              role="server"
              name="serverItf"/>

   <!-- the class implementing the interface ->
   <content class="multicusecase.impl.ServerImpl"/>

   <!-- the controller type->
   <controller desc="primitive"/>
</definition>
```

The file contains the definition of the server interface used for accepting new activation requests from outside. Note that no bindings need to declared since the wiring between the server interfaces and outer client interfaces have already been declared in the main description file `testcase.adl`. In the case of a primitive component, the tag `<content class=...>` must to be used to specify the Java class implementing the component business code.

2.3 Definition and implementation of a user-defined controller

The description ADL file `multicastComposite.fractal`, contains the declaration of the set of controllers to be associated to the composite component. By specifying the entry

```
<controller
desc=''/multicusecase/descriptor/AutonomicControllerMulticastItf.xml''>
```

the framework associates to the given component all the interfaces and the implementations of controllers it needs, as specified in the XML file.
In detail, the file `AutonomicControllerMulticastItf.xml` associates to the multicast controller of the composite component, the native controller MulticastControllerImpl.

The native version of the controller is in charge of applying a proper distribution policy *to each list* given as input task to the invoked multicast interface method it controls. As a consequence, the $1 : N$ communication pattern a multicast interface exploits, is strictly applied to the single list of tasks and consecutive method invocations on the same interface are independent the one with respect to the other, i.e. they are *stateless*. Nevertheless, in the perspective of implementing a stream parallel communication pattern, we need to alter the multicast interface behaviour in order to express a $1 : 1^*$ distribution (*one to one chosen in a set*, a.k.a. *unicast*) communication pattern, in which the destination of each task can be dynamically defined on the basis of interface *state*. The state may keep the trace of the distribution of previous items in the stream and the current status of the workers (sub-components). The decision process for the distribution of each task is a key issue for the definition and the exploitation of both the *functional replication* skeletons defined in D.NFCF.01 and future autonomic features. As a first approximation of a more flexible multicast controller, we provided a new implementation of this controller interface. In particular, the native version of the controller offers a distribution policy in which, for each method call, the distribution of the tasks in the input list always starts from the same component: in our version, the multicast controller records the last sub-component (i.e. worker) it sent a tasks and, at the next method call, it deliver the task to the successive one. The class implementing the new multicast controller is the `multicusecase/descriptor/RoundRobinPolicy.java`.
If you want to switch from one version of the multicast controller implementation, to the other one, you simply need to specify the actual implementation between the `<implementation>...</implementation>` tag related to the `MulticastController` interface.

```
<controller>
   <interface>
      org.objectweb.proactive.core.component.controller.MulticastController
   </interface>
   <implementation>
     multicusecase.descriptor.RoundRobinPolicy
   </implementation>
</controller>
```

Notice that this version of the multicast controller does not fully implement the unicast communication pattern we actually need for expressing stream parallelism, because we are not concretely able to overtake the fact that the multicast interface has been conceptually designed to support a list-based pattern of parallelism. This is why we needed to implement a user-defined proxy on top of a single server interface, as will explained in details in Sec. 4 and Sec. 5.

2.4 Implementation classes

The main class `Test.java` is available in the main package (`multicusecase`). The application invokes a number of method calls to the server interface of the `Producer` component `testConnectedServerMulticastItf`, by passing the number of tasks to be produced (i.e. the length of the list to be given as argument to the multicast interface) and the size of each tasks (lists of `WrappedInteger` values).

In between the activations of the producers, the autonomic controller (implemented by the class `AutonomicControllerMulticastItfImpl`) is asked to increase

the parallelism degree by invoking the operation `"increaseParallelDegree"` as a primitive one (passive level).

Each worker is implemented by the class `ServerImpl`, that implements the interface `ServerTestItf`. Such interface exposes the method

```
public void getService(WrappedInteger a);
```

since the multicast interface `MulticastTestItf` to which it is bound exposes the method

```
public void getService(List<WrappedInteger> a);
```

Note that returning a `void` type is the way we use to emulate a stream communication.

The service offered by the worker is a dummy function lasting a couple of seconds.

By running the application you will realize that if the tasks pushed into the farm (`long` values passed through the multicast interface method's input list) are less than the number of available workers, you will not be able to improve the overall service time. Increasing the parallelism degree won't help in getting a better performance. In fact, the distribution policy is very strict in this sense: each task *must* be a finite list of elements and these elements are distributed as a whole. Then, the availability of computing elements does not depend on the adaptive behavior of the component but on the structure of the single task. This is primarily why we worked on a way to adapt the component behavior by looking at the application's features instead of the single task structure (see Sec. 4 and 5). A consequence of the fact that a multicast interface works on a task base (i.e. defines its distribution policy on the basis of the single task structure) is that successive invocations of the interface methods will be serialized, loosing the ability to express parallelism among task, i.e. stream-parallel computations.

3 Implementing a synchronous farm using the Multicast interface

The following application is structurally quite similar to the previous one but in this case the method calls to the multicast interface are plain RPC method calls, i.e. they return a value to the caller. Moreover, before returning the service result to its caller (the `Producer`), each worker invokes a method call on a server interface belonging to a component (the `Collector`) outside the farm, thus augmenting the synchronicity level between sub-components.

3.1 Description of the application

The application is quite similar to the one described in Sec. 2.2, but the overall application is represented by three stages: the `Producer`, the composite component representing the farm and exposing the multicast interface and the `Collector`. The `Collector` is described by the `collector.fractal` file and exposes a server interface to which the farm sends all the results computed by the inner worker by invoking the proper `Collector`'s server interface.

The overall application is depicted in Fig.2

The farm has been detailed in Fig.3: in this version, the composite component exposes two interfaces, a multicast server interface distributing the work load to the inner sub-components (the *workers*) via a round-robin policy and a client interface to which all the sub-components will be bound to send their results to the collector component.

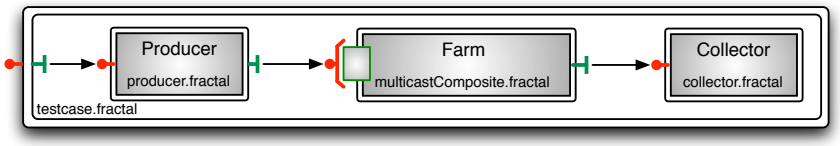

Figure 2: Application structure

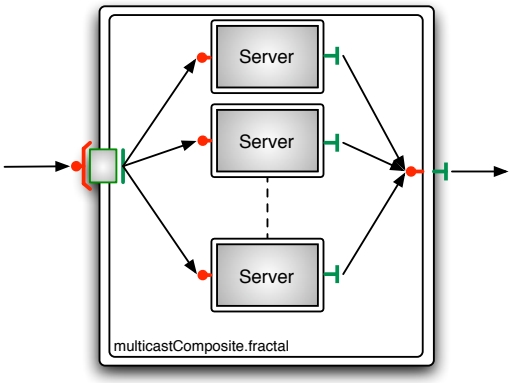

Figure 3: Structure of the farm component

We will detail the `testcase.fractal` component, just to clarify the concept.

```
<?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="multicusecaseSynch.adl.testcase">
     <interface signature="multicusecaseSynch.itf.Producer"
                role="server"
                name="runTestItf"/>

  <!-- the producer component -->
     <component name="tester"
                definition="multicusecaseSynch.adl.producer"/>

  <!-- second component: the farm -->
     <component name="multicastComposite"
                definition="multicusecaseSynch.adl.multicastComposite"/>

  <!-- the collector component -->
     <component name="collector"
                definition="multicusecaseSynch.adl.collector"/>

  <!-- the inner bindings -->
     <binding client="this.runTestItf"
              server="tester.runTestItf"/>
     <binding client="tester.clientItf"
              server="multicastComposite.multicastServerItf"/>
```

```
    <binding client="multicastComposite.CollectorItf"
             server="collector.runTestItf"/>

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

As you can see by comparing this file with the `testcase.fractal` presented in Sec. 1, the only major differences between these two versions are represented by the tags defining the collector component:

- a `<component ...>` tag for declaring its logic name and the .fractal file describing it

```
<component name="collector"
           definition="multicusecaseSynch.adl.collector"/>
```

- and a `<binding ...>` tag for binding the client interface belonging to the farm component to the collector's server interface.

```
<binding client="multicastComposite.CollectorItf"
             server="collector.runTestItf"/>
```

The collector component is a primitive component exposing a server interface. Its Fractal description file is given as follows (`collector.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="multicusecaseSynch.adl.collector">
  <interface signature="multicusecaseSynch.itf.CollectorItf"
             role="server"
             name="runTestItf"/>

  <content class="multicusecaseSynch.impl.CollectorImpl"/>
</definition>
```

3.2 Implementation classes

The existence of the collector implies the introduction of client interface on the component implementing a worker to be internally bound to the client interface provided by the farm. Thus, the `multicastComposite.fractal` definition file, exposes

- a multicast server interface as in Sec. 2.2

```
<interface signature="multicusecaseSynch.itf.MulticastTestItf"
           role="server"
           name="multicastServerItf"
           cardinality="multicast"/>
```

- and a client interface that will be bound to the collector's server interface `runServerItf`,

```
<interface signature="multicusecaseSynch.itf.CollectorItf"
           role="client"
           name="CollectorItf"
           cardinality="singleton"/>
```

as described in the `testcase.fractal` file. Note that the client interface is a `singleton` interface.

Each server component representing a farm is described in the file `server.fractal`, quite similar to the one detailed in Sec. 2.2 but also offering a client interface

```
<interface signature="multicusecaseSynch.itf.CollectorItf"
           role="client" name="CollectorItf" />
```

that will be bound in the farm component definition file to the farm's client interface.

```
<binding client="server0.CollectorItf" server="this.CollectorItf"/>
<binding client="server1.CollectorItf" server="this.CollectorItf"/>
<binding client="server2.CollectorItf" server="this.CollectorItf"/>
<binding client="server3.CollectorItf" server="this.CollectorItf"/>
<binding client="server4.CollectorItf" server="this.CollectorItf"/>
```

3.3 Observations

In both the synchronous and asynchronous versions of the application, we have a number of problems due to the emulation of a stream of tasks through a (bounded) set of structures that need a fixed amount of memory.
Generally speaking, we have had two kinds of problems:

- `OutOfMemoryException` raised each time we try to emulate a very long stream: this problem is probably due to the data structures (typically `ArrayList`) that ProActive run-time system allocates to manage futures and/or data returns. When the stream you want to emulate is too long, you may encounter this kind of exception.
- `NullPointerException` raised when we try to access to values not readily available: this problem is probably due to the way data are accessed before they are ready. The waiting behaviour of the read operations seems to be not always safe.

4 Implementing a farm by a user-defined proxy: passive level

The main goal of this application is to implement a farm in which a stream parallel computation is emulated by means of a method call to standard fractal/proactive interfaces that distributes the method calls, in a round-robin fashion, to a set of components acting as farm workers.

4.1 Structure of the package `passivefarm`

This package is composed by 4 sub-packages, as depicted in the table above, and one more file (`Main.java`) needed to launch the application and interact with it.

4.2 Description of the application

The application component (described in the `application.fractal` file and depicted in the 5 figure) exposes a server interface offering a service: each invocation to such server interface activates a `Tester` (described by the `tester.fractal` file) component that is in charge of generating streams of requests to be sent to the farm. Each stream consists in a certain number of functional method invocations.

Name of the sub-package	Description of the sub-package
`adl`	contains the ADL descriptors configuring the application
`deployment`	contains the deployment XML file
`impl`	contains the user implementation class of the application
`itf`	contains the user interface class of the application

Table 2: Composition of the `passivefarm` package

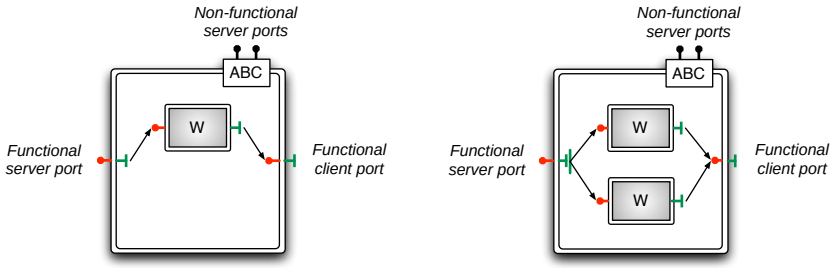

Figure 4: Passive Farm structure

The farm component (depicted in Fig. 4) consists in a composite component which structure is described in the `farm.fractal` file; it exposes a server interfaces offering a service and contains a sub-component which can be replicated on-demand by the Autonomic Behavior Controller (ABC). After their creation, the sub-component duplicates are scheduled on virtual nodes whose names follow this template: `<original-subcomponent-name>-vn-<integer-number>`. The ABC assumes to find a deployment folder (called "Deployment") outside of the main package (using the File System metaphor, the main application package and the deployment folder have the same parent), inside the folder the ABC looks for the file "`deployment-descriptor.xml`" and parse it searching virtual nodes matching the template stated above. The ABC is present in the composite component membrane even if the `farm.fractal` file does not directly specify a customized controller description file because it extends the `AutonomicControllerForFarm.fractal` ADL file contained in the package

`org.ercim.gridcomp.component.autonomic.controller.adl`

where the information about the embedding of ABC is specified.

The method calls invoked on the farm component server interface are scheduled among the available workers exploiting the round-robin policy. Each call is en-queued into the local queue of the assigned worker. Each worker executes each call en-queued inside its local queue eventually returning the result of the method execution if the method return type is not void.

To clarify what we described, it can be useful to have a look at the ADL files cited above. First of all the `application.fractal` one:

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

<definition name="passivefarm.adl.application">

  <interface signature="passivefarm.itfs.Tester"
```

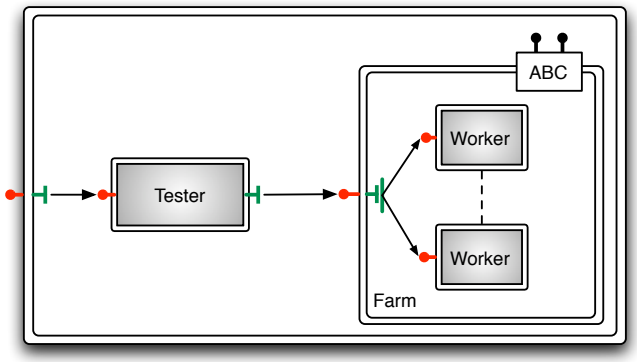

Figure 5: Application with the passive farm

```
    role="server" name="runTestItf"/>

  <!-- first stage: the tester -->
  <component name="tester" definition="passivefarm.adl.tester"/>

  <!-- second stage: the farm -->
  <component name="farm" definition="passivefarm.adl.farm"/>

  <!-- internal bindings -->
  <binding client="this.runTestItf" server="tester.runTestItf"/>
  <binding client="tester.collectiveClientItf" server="farm.serverItf"/>

  <controller desc="composite"/>

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

The `definition` tag, as usual, indicates the name of ADL file and bounds the area in which the component definition tags must be specified. The first line inside such area describes the external interface of the application component, its server role, its signature and its name. Then two `component` tags are present, the former refers to the `tester` component and the latter to the `farm` one. After, the binding among the two internal components and the `application` one (clearly represented by the `this` keyword) are specified. In the last part of the file are inserted the description of component controllers and the information about virtual node for the component deployment.

The `tester` is a primitive component, hence its ADL is quite simple:

```
<?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="passivefarm.adl.tester">

  <interface signature="passivefarm.itfs.Tester" role="server"
                name="runTestItf"/>
  <interface signature="passivefarm.itfs.ServerItf" role="client"
                name="collectiveClientItf"/>
```

```
  <content class="passivefarm.impls.TesterImpl"/>

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

Indeed, inside the definition part there are only the declaration of component external interfaces, the name of the java class implementing the component and eventually the deployment information. Writing the ADL of the `farm` component is still very simple, nevertheless it needs to pay more attention.

```
<?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="passivefarm.adl.farm"
extends="
org.ercim.gridcomp.component.autonomic.controller.adl.\
AutonomicControllerForFarm">

  <interface  name="serverItf" role="server" contingency="mandatory"
              signature="passivefarm.itfs.ServerItf" />

  <component name="server"
             definition="passivefarm.adl.server"/>

  <binding client="this.serverItf"
           server="server.serverItf"/>

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

In this case the `definition` tag specifies an extends relationship between the `farm` ADL file and the `AutonomicControllerForFarm` one, which has been designed to ease the embedding of ABC inside components. It permits to avoid the controller declaration inside the `farm` ADL file separating the architecture description issues required for the autonomicity from the functional one. The following parts of the ADL file are pretty trivial: the definition of an interface with the same role, signature and contingency of the one exposed by the internal component, the embedding of the replicable component (called `server` in this example), the binding between the `farm` and `server` component and the deployment information.

The last ADL file defined in this example is `server.factal`. It describes the structure of the primitive component inserted into the farm composite component to be duplicated on demand by the ABC. In particular, the nature of its external interface, the class containing the component "business–code" and deployment information:

```
<?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="passivefarm.adl.server">
  <interface signature="passivefarm.itfs.ServerItf"
    role="server" name="serverItf"/>

  <content class="passivefarm.impls.ServerImpl"/>

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

4.3 Implementation details

This component application is instantiated and executed by the `Main` class, implemented by the `Main.java` file present in the main package (`passivefarm`). Once started the `Main` class exploits a number of method calls to the server interface of the `Tester` component (`runTestItf`), passing, as arguments, the number of tasks to be produced (i.e. the number of method calls to invoke on farm server interface) and the size of each task (lists of `Integer` values).

Between one activation and the next one of the `Tester`, the autonomic behavior controller (implemented by the class `AutonomicControllerForFarmImpl`) of the farm component is asked to increase (or decrease) the parallelism degree (number of internal sub-components containing the "business–code" and acting as workers). The operations "`increaseParallelDegree`" and "`decreaseParallelDegree`") are called through the invocation of the `execOperation` method, which can be manually triggered by the user via the the "+" and "-" keys (respectively), followed by "`Enter`" key on the same terminal in which the application has been launched.

The farm internal component(s) "business–logic" is implemented by the class `ServerImpl`, that implements the interface `ServerItf`. Such interface exposes the method

```
IntWrapper compute(Vector<Integer> a)
```

Taking a vector of `Integer` as input, executing a dummy iterative function for each `Integer` element and eventually returning a result that is a combination of the results obtained for each `Integer` computed.

4.4 Summarizing the general rules

In the previous sections we described in deep an applicative testbed. It is made by two main components, a tester and a farm. The farm parallelism degree can be increased or decreased on demand simply invoking a non-functional method on an ABC able to reconfigure the farm. The non-functional methods implemented by the ABC are consistent with the GCM autonomic specification:

- **String[] listAutonomicOperations():** method to obtain the list of non-functional operations provided by the AB controller instance upon which it is invoked.
- **any execOperation(String op, params...):** method to execute the non-functional operation whose name is specified as first argument. The other arguments consist of operation input parameters.

The farm is equipped with an ABC instance (`AutonomicControllerForFarm1`) offering three non-functional operations:

- `increaseParallelDegree`;
- `decreaseParallelDegree`;
- `JobEnqueuingTime`.

This last one can be used to monitor the average time each component request requires to be completed (completion time): the en-queuing time added to the computing time. To use a farm in place of a primitive component in a Fractal/ProActive application is quite easy, and can be done simply writing an additional ADL file defining the farm and changing the ADL files referring to the replaced component ADL substituting its name with the farm ADL name. To write the farm file the programmers are required to follow simple rules:

- the `definition` tag must extends the `AutonomicControllerForFarm` definition
- the farm component must contain only one sub-component: the one to be replicated
- the farm component must expose all the interfaces exposed by the internal sub-component

This is what is needed to have a farm component in place of a primitive component able to reconfigure itself. Nevertheless the reconfiguration activity must be triggered by an external entity (passive farm). In the next section will be described how to implement a farm able to trigger a reconfiguration action on itself given a specific non-functional contract (active farm).

5 Implementing a farm by a user-defined proxy: active autonomic control

The application presented in this section has a structure similar to the one presented in the previous section, the main difference regards the way the farm reconfigures itself. Indeed, if in the previous section is described how to replace a component with a farm able to increase or decrease its parallelism degree on demand, in this section is described how to replace a component with a self-reconfiguring farm.

5.1 Structure of the package `activefarm`

The structure of the application described here has a few differences w.r.t. the one introduced for the passive farm. In this case the package is composed by 5 sub-packages, as depicted in the table above, and one more file (`Main.java`) needed to launch the application and interact with it.

Name of the sub-package	**Description of the sub-package**
`adl`	contains the ADL descriptors configuring the application
`autonomics_element`	contains the implementation of the manager for this application
`deployment`	contains the deployment XML file
`impl`	contains the user implementation class of the application
`itf`	contains the user interface class of the application

Table 3: Composition of the `activefarm` package

5.2 Description of the application

The application component (described in the `application.fractal` file) contains two components: `client` (described by the `client.fractal` file) and `stage` (described by the `composite_stage.fractal` file). `client` component is in charge of generating a stream of requests to be sent to the `stage` one (the farm component); The stream consists of functional method calls. The farm (its structure is represented in Fig. 6) is represented by the composite component described in the `composite_stage.fractal` file:

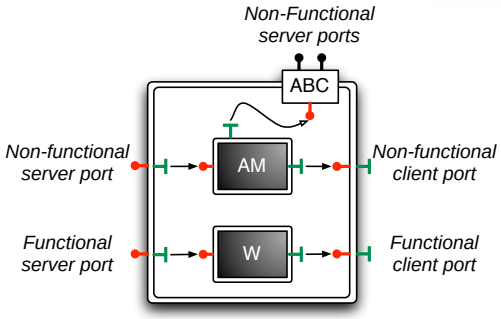

Figure 6: Active Farm structure

```
<?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="activefarm.adl.composite_stage"
         extends="activefarm.adl.customized_farm_with_manager">

  <interface name = "server-compute"
             role = "server"
             contingency = "optional"
     signature = "activefarm.itfs.Compute"/>

  <component name="innerStage"
             definition="activefarm.adl.server" />

  <binding client="this.server-compute"
           server="innerStage.server-compute" />

</definition>
```

Similarly to the application presented in Sec. 4 the farm component is implementable simply declaring for it the same interfaces the internal sub-component exposes, embedding the proper "business–logic" component, performing internal binding and declaring the ADL definition as an extension of a certain other ADL definition. In the previous application the definition to be extended was fixed and part of the autonomic framework (`AutonomicControllerForFarm`) whereas in this case it must be properly written or at least customized. That's because it contains the reference to the ADL defining the autonomic manager, typically implemented by the same programmer implementing the application. In this example the ADL extended is `activefarm.adl.customized_farm_with_manager`:

```
 <?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="activefarm.adl.customized_farm_with_manager">
<component name="automan"
definition="activefarm.adl.customized_manager"/>

  <binding client="automan.client-autonomic-controller"
server="this.autonomic-controller"/> <controller
```

```
desc="/org/ercim/gridcomp/component/autonomic/controller/
config/AutonomicControllerForFarm.xml"/>

</definition>
```

It contains the declaration of the autonomic manager component (`automan`), the internal bindings and the controllers description. As we stated before this file is not fixed and it is not provided with the autonomic framework however its implementation is quite simple and, roughly speaking, the programmer can reuse the same file written once and for all simply replacing the name of the manager definition file with the one defining the manager the programmer wants to use.

The ADL file defining the autonomic manager of this example is the following:

```
 <?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="activefarm.adl.customized_manager" extends="
org.ercim.gridcomp.component.autonomic.manager.adl.\
AbstractAutonomicManager">

<content class=
"activefarm.autonomic_elements.FarmificationAutonomicManager"/>

</definition>
```

The structure of this file is quite simple: a definition tag extending the ADL file `AbstractAutonomicManager` and the declaration of the class name containing the autonomic manager implementation.

The `AbstractAutonomicManager` ADL file defines all the interfaces an autonomic manager must have, namely a `commit-contract` server interface, two `raise-violation` interface (an internal one with a server role and an external one with a client role) and a `client-autonomic-controller` interface bound to the autonomic controller of the composite component containing the autonomic manager.

```
<?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.ercim.gridcomp.component.autonomic.manager.adl.\
AbstractAutonomicManager">

  <interface name = "commit-contract"
             role = "server"
             contingency = "option"
signature = "org.ercim.gridcomp.component.autonomic.manager.itf.\
AutonomicServerManager"/>

  <interface name = "raise-violation"
             role = "client" contingency = "optional"
signature = "org.ercim.gridcomp.component.autonomic.manager.itf.\
AutonomicClientManager"/>


  <interface name = "internal-raise-violation"
             role = "server"
```

```
            contingency = "optional"
signature = "org.ercim.gridcomp.component.autonomic.manager.itf.\
AutonomicClientManager"/>


  <interface name = "client-autonomic-controller"
            role = "client"
            contingency = "optional"
signature = "org.ercim.gridcomp.component.autonomic.controller.itf.\
AutonomicController"/>


  <virtual-node name="autonomic-manager-node"
                cardinality="single"/>

</definition>
```

The autonomic manager for the active farm of this example application is implemented through the "`FarmificationAutonomicManager`" class, that extends the "`AbstractAutonomicManager`" one and implements the abstract method

```
protected void enforcePerformanceContract(String qosContract)
```

that receives a performance contract as input and try to enforce it acting on the ABC of the farm component for adjusting the parallel degree. Currently we are investigating several possibilities about the structure of the performance contract. Hence, in the current implementation of the autonomic support, the parsing and the management of the qosContract should be implemented by the AM programmers.

All these steps of ADL files instrumentation, are required to insert the autonomic manager and the ABC into the composite component and its membrane and creating a binding between them.

5.3 The implementation class

The main class `Main.java` is available in the main package (`activefarm`). It instantiate the application and start its life-cycle. The client component of the application exploits a number of method calls to the server interface of the `stage` component `server-compute`.

The autonomic manager (class `AutonomicControllerForFarmImpl`) of the farm component monitors the average en-queuing time of the task computed by the farm and asks ABC to increase (or decrease) the parallelism degree (number of available workers) when the performance contract is not obeyed.

To measure the en-queuing time, the manager must activate the monitoring support, in fact during its initialization call the "`JobEnqueuingTime`" non-functional operation with `new Object[]{new Boolean(true)}` as parameter. For each observed method, in order to obtain the average completion time of the last 5 method calls the manager invokes "`JobEnqueuingTime`" non-functional operation with "`new Object[0]{ObservedMethodName}`" as parameter, where "ObservedMethodName" is the name of the method the manager wants to obtain completion time. Each worker is implemented by the class `Server`, that implements the interface `Compute`. Such interface exposes the method

```
GenericTypeWrapper compute(Integer i)
```

The service offered by the worker is a dummy iterative function.

5.4 Summarizing the general rules

In this section we have described how to implement a component application in which in place of a primitive component the programmer needs an autonomic farm able to increase or decrease its parallelism degree automatically. In particular has been stated that the programmer is in charge of performing the following steps:

- write your own autonomic manager such that it extends the abstract class `AbstractAutonomicManager`
- customize the ADL file describing the manager
- customize the ADL file describing the farm structure
- write a ADL file extending the ADL file describing the farm structure, embedding the "business–logic" component and declaring the same external interfaces of the contained sub-component

The last step is the same required for the passive version of the farm (see Sec. 4). Instead, the first three steps are specific of this active version.

6 How to compile

All the experiments have been developed under ProActive 3.2.. The compilation process requires a compliance level to Java 1.5 and you need to include a set of java packages in the classpath: as an example, if you use Eclipse, you should configure your Java Build Path as depicted in Fig. 7.

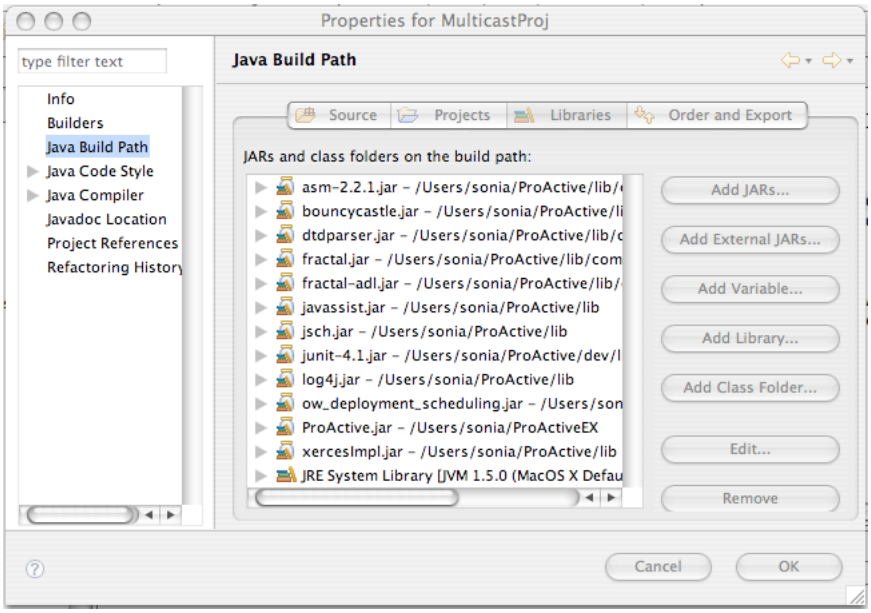

Figure 7: Libraries required for compiling the application

7 How to run the applications

In order to start the application, type of your Eclipse `Run` window the name of the main class of the project.

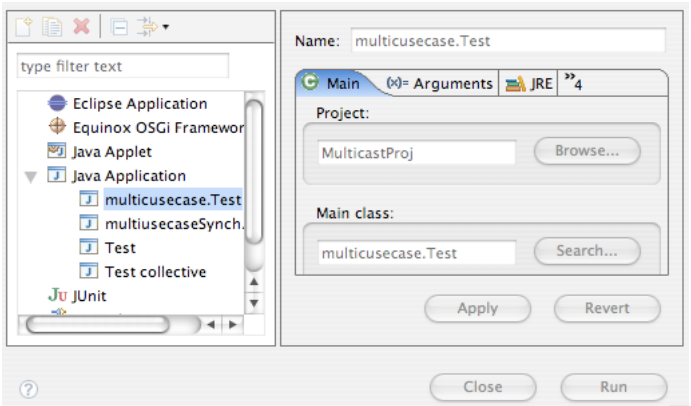

Figure 8: `Run` Eclipse window configured to run the application

Moreover, remember to configure the arguments needed by the JVM in the `Arguments` tab (see Fig. 9).

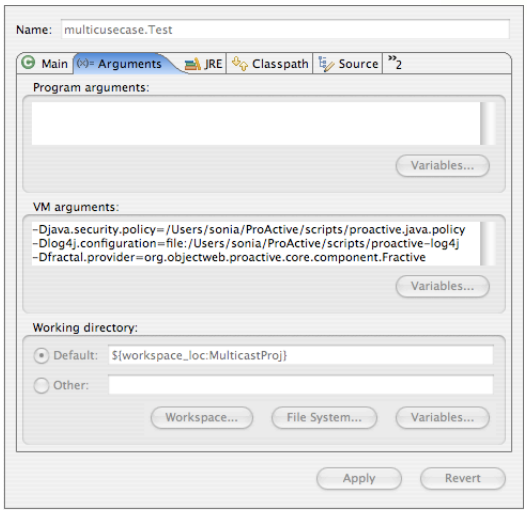

Figure 9: Arguments to configure the JVM

References

[1] M. Aldinucci, S. Campa, P. Dazzi, and N. Tonellotto. *D.NFCF.01 – Non functional component subsystem architectural design*. GridCOMP STREP deliverable D.NFCF.01, June 2007.

[2] E. Bruneton. *Developing with Fractal*. The Object Web Consortium, Feb. 2004. `http://fractal.objectweb.org/tutorials/fractal/index.html`.

[3] T. O. W. Consortium. ProActive official homepage: `http://www-sop.inria.fr/oasis/ProActive/index.ph`.

[4] J. S. E. Bruneton, T. Coupaye. *The Fractal Component Model*. The Object Web Consortium, Feb. 2004. `http://fractal.objectweb.org/specification/index.html`.

[5] C. N. of Excellence. *Deliverable D.PM.04 - Basic Features of the Grid Component Model*. The Object Web Consortium, Sept. 2006.

[6] O. R. Team. *ProActive Manual v.3.2*. The Object Web Consortium, Apr. 2007.