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.04 – NFCF prototype and early documentation

Due date of deliverable: May 31^{th}, 2008

Actual submission date: June 25, 2008

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		
PU	Public	PU

Keyword list: autonomic management, component controller, GCM, behavioural skeleton, task farm, data parallel
Responsible Partner: UNIPI

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	6 June 2008	Draft	Sonia CAMPA
1	8 June 2008	Draft	Marco ALDINUCCI
2	9 June 2008	Draft	Marco ALDINUCCI
3	24 Jun 2008	Final	Marco ALDINUCCI

Deliverable manager

- Marco Aldinucci, UNIPI

List of Contributors

- Sonia Campa, UNIPI
- Patrizio Dazzi, ISTI-CNR
- Nicola Tonellotto, ISTI-CNR
- Giorgio Zoppi, UNIPI

List of Evaluators

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

Executive Summary The D.NFCF.01 deliverable [3] of the GridCOMP project, provides an architectural specification of a GCM autonomic component [4]. A GCM autonomic component is a component exploiting 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 behaviour of the component is leaded by a manager who takes care of planning and taking adaptive decisions.

The current deliverable is intended to be a tutorial concerning the step-by-step usage of the second year prototype version of a GCM autonomic components implementation we have developed in the GridCOMP project context, focusing on behavioural skeleton. Behavioural skeletons abstract component self-management in component-based design as design patterns abstract class design in classic OO development. Thus, behavioural skeleton describe recurring patterns of component assemblies that can be equipped with current and effective management strategies (as the ones defined in D.NFCF.03 to which this document is related) with respect to a given management goal provided as a Quality of Service (QoS) contract.

All the examples and use-cases described in this tutorial are used to illustrate the usage of the autonomic management API. The examples are a part of the source code distributed as a zip archive with the current document.

Contents

1 Introduction

The objective of this tutorial is to illustrate how to design, to compile and run an autonomic application defined as an assembly of autonomic components. The underlying programming model is the one provided by the Grid Component Model (GCM) [4], which design has been inspired by the Fractal component model [6]. The reference implementation of GCM considered in the presented examples is developed on top of the ProActive middleware ver. 3.9 [7].

The tutorial focuses particularly on the usage of behavioural skeletons, which are high-order, parametric component assemblies. These skeletons can be equipped with specialised management strategies driven by pre-defined performance goals [1, 2].

In this document, we discuss the *functional replication* skeleton family, which enables the application designer to easily exploit a dynamically variable number of replicas of a guest (primitive or composite) component. The `farm` and `data parallel` skeletons are instances of the functional replication skeleton family.

The `farm` skeleton realises the parallel computation of independent tasks parallel pattern, whereas the `data parallel` skeleton exploit the homonyms pattern. These skeletons are realised in GCM as particular composite components that exploit a set of *autonomic controllers* implementing the protocols for the correct dynamic adaptation of the skeleton (e.g. life cycle, content, binding, etc.). In particular, adaptation operations are exposed by the *Autonomic Behaviour Controller* (ABC) of the component [2]. These operations can be used to dynamically steer the component behaviour (i.e. passive autonomicity).

Behavioural skeletons (a.k.a. *BeSke*) are obtained by equipping skeletons with an *autonomic manager* (see Fig. 1) exploiting a proper management goal. Truly autonomic components can be obtained as instances of the behavioural skeletons (i.e. active autonomicity).

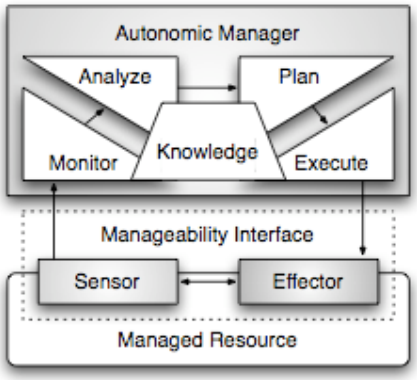

Figure 1: Autonomic computing control loop

The autonomic management happens according to four successive phases. The *monitor* phase provides mechanisms to collect, aggregate, filter and correlate details from managed resource (memory, work load, remote resources, etc.); the *analyse* phase provides mechanisms to observe and analyse behavioural or structural changes (for example a change in the work load of a given machine involved in the application execution); the *plan* phase provides mechanisms to create or select a procedure to change the current status of the observed resources; the *execute* phase applies such procedures by triggering operations directly at the passive level. All these phases are conditioned by a (set of) goal that a user could specify by means of a Quality of Service (QoS) contract.

In this tutorial we will show how farm and data parallel skeleton can be instanced in a set of use-cases both at the passive and the active level by providing to the manager a set of goals defined by the user in the form of a contract file.

1.1 Structure of the code

The framework code is distributed as a zipped archive containing four main directories:

- `GridComp-Core`: contains all the packages representing the core of the frameworks: interfaces and abstract classes of controller, managers, autonomic operation, etc.
- `GridComp-Manager`: contains all the classes needed for the implementation of the autonomic manager
- `GridComp-Farm`: contains all the interfaces, abstract and implementation classes related to the farm behavioural skeleton and its manager implementation
- `GridComp-Map`: contains all the interfaces, abstract and implementation classes related to the data-parallel behavioural skeleton and its manager implementation

All these directories are linked to the `GridComp-Libs` directory collecting all the ProActive, Fractal and Java library needed to compile and run the code.
The use-cases are represented by the following directories code:

- `GridComp-Example`: collects a set of applications exploiting both active and passive autonomicity in the farm and data-parallel context. The applications are:
  - Mandelbrot set: it's the classic Mandelbrot application exploiting pure task parallelism in which a set of independent tasks are computed at different level of complexity by a set of processing elements represented by the workers of a farm
  - π calculus: as the previous example, this application adopts a task parallelism patterns but the structure of the whole application is slightly different
  - stateful data parallelism: it is a data parallel application in which the content of the input data is partitioned and computed by the set of worker components, which may have an internal state provided the worker components expose a method for importing and exporting the internal state. In case of skeleton re-configuration, the state is re-distributed under the coordination of the manager.
  - stateless data parallelism: the example is mockup of the IBM use case [8]. Here, worker components are assumed to be stateless (or including a not disclosed state), thus in the case of re-configuration the manager does not provide any automatic way to redistribute the worker state. In the case the worker components exhibit a not disclosed state, the redistribution of state can be managed by way of functional interfaces, i.e. providing components with the proper functional ports to manage it. This management should explicitely programmed by the application designer by way of the state management interfaces.
- `GridComp-Dicom-Use-Case`: it's a real application based on the recognition of breast x-rays images.

Summarising, the structure of the archive can be seen in Table 1

2 Getting started

2.1 Platform requirements

In order to compile and run the framework you will need Java 1.5 or upper versions. ProActive 3.9 and all the auxiliary libraries are provided within the zipped archive file.

GridComp-Core	implementation of the framework core
GridComp-Manager	abstract classes, interfaces and ADL files for implementing the manager
GridComp-Farm	abstract classes, interfaces and ADL files for implementing the farm behavioural skeleton; prototype implementation of a farm behavioural skeleton supporting active autonomicity
GridComp-Map	abstract classes, interfaces and ADL file for implementing the data parallel behavioural skeleton; prototype implementation of a data parallel behavioural skeleton supporting active autonomicity
GridComp-Example – **Mandelbrot** – **pi** – **stateless data-parallel** – **stateful data-parallel**	 evaluation of the Mandelbrot set (autonomic farm) evaluation of the π approximation (autonomic farm) data parallel use-case supporting functional reconfiguration data parallel use-case with functional reconfiguration
GridComp-Dicom	Dicom use case example for the recognition of X-ray images

Table 1: Content of the framework distribution.

2.2 Installing the code

In order to install the code, you need to enter your working directory and digit:

```
~ > tar -xzvf gridcomp.tar.gz
~ > cd gridcomp
~/gridcomp > ls
GridComp-Core            GridComp-Farm            build.xml
GridComp-DataParallel    GridComp-Libs            commons.xml
GridComp-Dicom-UseCase   GridComp-Manager         distribution
GridComp-Examples        bin                      jarlib
```

Opening the archive will lead to the creation into the working directory of a directory called `gridcomp` containing the framework implementation directory and other files for supporting compilation and running.

2.3 Compiling the code

The compilation process is supported by the ant compiling system. Thus, in order to compile the framework, it suffices to type

```
~/gridcomp > ant
```

which will invoke the compilation steps programmed into the `build.xml` file. Before launching the compilation, make sure that your `JAVA_HOME` environment variable is correctly set.

If the user wants to compile one of our example application only, he/she need to enter the directory `GridComp-Examples` and to call the compilation process from the related ant build file, thus:

```
~\gridcomp > cd GridComp-Examples
~\gridcomp\GridComp-Examples > ant build-passive-mandelbrot
```

However, when invoked without parameters, ant will compile all the provided example, by default[1].

[1] In order to know which target type to compile and launch the other examples, take a look to the build.xml file in the `GridComp-Example` directory.

2.4 Running your first example

Once the user has compiled the code, she/he can simply invoke its execution through the ant mechanism. We provide two type of deployment descriptors for driving the execution phase: the `local-deployment.xml` file that allows to deploy the application on your local machine and the `distrib-deployment.xml` descriptor that is configured to deploy the application on a cluster. In both cases, you simply need to invoke the execution process on the target representing the application you are interested in. For example, in order to launch the evaluation of the Mandelbrot set, type:

```
~/gridcomp/GridComp-Example > ant local-deploy-mandelbrot
```

3 Programming rationale behind behavioural skeletons

Behavioural skeletons represent an easy and abstract way to express well-known parallel computation patterns by using pre-defined composite component. The main advantage of using such kind of component provided with specific behaviour is that the knowledge behind their behaviour specification allows to pre-define autonomic operation in order to adapt the component structure and/or behaviour depending on events triggered by the external or internal environment.
All our behavioural skeletons come along with a pre-defined set of autonomic controllers ensuring the passive level of autonomicity and a version of the autonomic manager related to the specific skeleton. The manager is a component with dedicated functions: it implements the autonomic cycle (see Fig.1) by using *when* <*condition*> *then* <*action*> rules while the application runs. Its role is to periodically evaluate specific monitored conditions and, if one or more holds, to take the specific autonomic reaction related to the condition occurred.

The type of information the manager has to check is provided through the quality of service (QoS) contract. The contract is a set of requirements that the user needs to be satisfied during the application lifetime and can be performance requirements as well as other qualitative and/or quantitative information about resource consumption, data sizes, network bandwidth or latency etc.

The contract is represented (and evaluated) in the manager as a list of JBoss rules [5]. In fact, the contract is a JBoss file collecting a set of *when* <*condition*> *then* <*action*> rules: each `AutonomicLoopCycle` seconds the manager queries the set of rules and executes the actions related to those matching *condition*. An example of a QoS contract is given below:

```
package gridcomp.manager
import gridcomp.manager.beans.*;
import gridcomp.manager.operations.*;
import gridcomp.operation.*;
import gridcomp.manager.map.impl.PartitionSizeBean;

[methodMonitor="searchMatch"]
rule "CheckHigherBound"
        when
            $arrivalBean : PartitionSizeBean(value >=10)
        then
            $arrivalBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
 end
[methodMonitor="getService"]
rule "CheckLowerBound"
        when
            $arrivalBean : PartitionSizeBean(value < 9)
        then
            $arrivalBean.fireOperation(ManagerOperation.REMOVE_EXECUTOR);
 end
```

In this contract (see `gridcomp/example/<example_name>/impl/rules.drl`), the manager is asked to monitor two methods in the interface of the component it controls, `searchMatch` and

`getService`. In other words, the manager checks if any of the rules in the QoS contract applies each time one of the methods marked in the contract are called. If the condition holds, the rule is executed. In this example, both rules require that the same Java Bean `PartitionSizeBean` is executed and the result stored in the variable `value` is checked for the given condition. If `value` is higher or equal to 10, then the autonomic operation `ADD_EXECUTOR`, belonging to the range of operations provided by the manager, will be invoked; else, the `REMOVE_EXECUTOR` will be executed.

The application developer can exploit the manager functionality by simply writing a QoS contract as a list of JBoss rules. This requires the knowledge of adaptation operations and monitor variables exposed by the particular component. Notice however that, if the component is an instance of a behavioural skeleton (*BeSke*), these operations and variables are typically pre-defined (and documented) as part of the *BeSke* definition.

The set of autonomic operations currently provided are listed in `gridcomp.manager.operation` and are:

- `ADD_EXECUTOR`: adds a new (local or remote) executor to the skeleton configuration (all *BeSke*).
- `REMOVE_EXECUTOR`: removes a (local or remote) executor from the skeleton configuration (all *BeSke*).
- `BALANCE_LOAD`: balances the load of tasks to be executed by a set of executors by reassigning them with respect to the queue distribution defined by the ProActive framework (`farm` only).
- `RAISE_VIOLATION`: signals that a violation contract occurred (all *BeSke*).
- `SETUP`: defines new configuration features of the skeleton (all *BeSke*).
- `GET_EXECUTORS`: provides the number of executors currently involved in the computation (all *BeSke*).

4 Implementing a data parallel behavioural skeleton

In this section we will show how an application exploiting a data parallelism pattern can be programmed with GCM behavioural skeleton into the framework. We will detail two cases: the first relates to a data parallel application in which the reconfiguration (i.e. the adding and removing of components as processing elements of the behavioural skeleton) is completely driven by the autonomic manager without any kind of human intervention. The second case relates to a data parallel skeleton whose processing elements are stateless components. In case of a reconfiguration, the redistribution of the input data has to be explicitly treated by the user.

4.1 Stateful data parallel skeleton

All the code referenced in the sequel belongs to the package `gridcomp.example.statefulmap.*`, if not differently specified. Moreover, the application implements the contract file `rules.drl` (see Section 3) defined in `gridcomp.example.statefulmap.impl`.

4.1.1 Semantics

The stateful data parallel behavioural skeleton is depicted in Fig. 2 The composite component representing the skeleton has been designed focusing on the following requirements:

- all the *provide* ports of the composite *must* be multicast ports

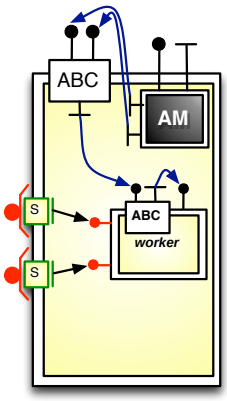

Figure 2: Stateful data-parallel *BeSke* structure.

- the user interfaces that define the signature of each multicast port can adopt all the distribution policies provided by ProActive (`BROADCAST`, `ONE_TO_ONE`, `ROUND_ROBIN`). However, in order to define a port exploiting the pure data parallel behaviour (i.e. the input task is split among the available processing elements of the skeleton), the port must be configured to adopt the `gridcomp.map.port.MapDispatch` as distribution policy.
- all the multicast port belonging to the composite are managed by the manager and the autonomic controller

In the following section, we will show how such composite can be used within the user code.

4.1.2 The main application structure

In the stateful example, the main application instantiates the composite by the following piece of code:

```
Map ctx = new HashMap();
String root;
root = new String("gridcomp.example.statefulmap.adl.testcase");
cxt.put("worker", "gridcomp.example.statefulmap.adl.worker");
cxt.put("rulespath",
  "../GridComp-Examples/src/gridcomp/example/statefulmap/impl/rules.drl");
Component testcase = (Component) f.newComponent(root,ctx);
```

A `Map` object is instanced in order to provide two arguments to the loading component engine of ProActive: the arguments are respectively

- the name in the package of the ADL file representing the worker (i.e. the component working on a single partition of the partitioned state)
- the path to the user QoS contract in the file system

Both the arguments are given as input to the component engine of ProActive together with the name of the ADL file representing the application component.

The ADL file representing the application component is defined 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="gridcomp.example.statefulmap.adl.testcase"
   extends="gridcomp.CompositeController"
   arguments="worker,rulespath">

<interface signature="gridcomp.map.port.MulticastTestItf1"
   role="server"
   name="runTestItf-01"/>

<interface signature="gridcomp.map.port.MulticastTestItf2"
   role="server"
   name="runTestItf-02" />

<component name="multicastComposite"
   definition="gridcomp.map.doubleMulticast(${worker},${rulespath})"/>

<binding client="this.runTestItf-01"
         server="multicastComposite.multicastServerItf-01"/>
<binding client="this.runTestItf-02"
         server="multicastComposite.multicastServerItf-02"/>
<controller desc="composite"/>
</definition>
```

The application is a composite extending the `gridcomp.CompositeController` composite: such extension allows the user to automatically configure a set of controllers and interceptors of GCM composite component. The composite encapsulates the data parallel behavioural skeleton (`multicastComposite` component) that is directly bound to the two input interfaces provided by the application component as its "entry-point". In other words, the application provides the same server port provided by the data parallel behavioural skeleton. The skeleton is instanced passing it two arguments: the name of the ADL file of the worker and the path to the QoS contract. These arguments are instantiated at running time by the component creation engine and the ADL arguments mechanism.

The data parallel skeleton In this example, the user instantiates the `gridcomp.map.doubleMulticast` version of the behavioural skeleton that exploits the following features (see fig. 2):

- the skeleton provides two (multicast) server ports
- the signatures are defined by the `gridcomp.map.port.MulticastTestItf1` as well as the `gridcomp.map.port.MulticastTestItf2` classes, respectively
- the first port exploits a pure data parallel semantics, i.e. it receives a list of items as inputs, splits it in a number of partition equal to the number of available processing elements (workers) and assigns a partition to each processing element; we will refer it as an *initialisation* port
- the second port exploits the BROADCAST distribution policy, as defined by the ProActive run-time system.
- the results are collected in an RMI style by the server port, themselves.
- as mentioned above, the skeleton is parametric with respect to the name of the ADL file describing an inner worker (the set of components hosted by the composite data parallel component) and the pathname of the QoS contract.

The processing element (worker) The second ADL file the user must provide is the one describing the component that will actually instance the data parallel skeleton working on a slice of the partitioned input data. In `gridcomp.example.stateful.adl.worker` the user can access the descriptive ADL file that appears 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="gridcomp.example.statefulmap.adl.worker"
extends="gridcomp.map.controller.DPWorkerController">

   <interface signature="gridcomp.map.controller.operation.ReconfigSupport"
              role="server"
              name="reconfig"/>
   <interface signature="gridcomp.example.statefulmap.impl.ServerTestItf1"
              role="server"
              name="serverItf-worker-01"/>
   <interface signature="gridcomp.example.statefulmap.impl.ServerTestItf2"
              role="server"
              name="serverItf-worker-02"/>
   <content class="gridcomp.example.statefulmap.impl.ServerImpl"/>

</definition>
```

Since users want to exploits the autonomic reconfiguration features offered by the data parallel skeleton, their worker must provide the autonomic support. In this sense, they simply need to define as worker a component that extends the `grid.map.controller.DPWorkerController`: such extension enriches the component definition with a set of pre-defined controllers and interceptors that provide the passive autonomicity level, at least, and all the needed hooks for the skeleton manager to interact with the worker.

4.1.3 The reconfiguration process

The reconfiguration is a three-steps process:

1. as first step, the partition already assigned to the current workers are collected at passive level
2. the reconfiguration proceeds with adding or removing a worker
3. as last step, the state collapsed in step 1 is partitioned and the partitions are distributed again among the new set of workers

However, since the state is part of the business code, the user is in charge of "instructing" the manager about how to serialise and de-serialise the state, after that the initialisation port has distributed it among the processing elements of the data parallel skeleton. The information about serialisation/de-serialisation is provided by the user implementing the

`gridcomp.map.controller.operation.ReconfigSupport`

interface as part of the data parallel skeleton worker.

```
package gridcomp.map.controller.operation;
import java.util.List; import gridcomp.map.Task;

public interface ReconfigSupport {
   public static final String CONTROLLER_NAME = "dpworker-controller";
   public void setStatus(List<Task> tsl);
   public List<Task> provideStatus();
}
```

Such interface requires the implementation of two methods:

- `set_status(List <Task> tsl)`: represents the de-serialisation process and is activated when, after a reconfiguration, the worker receives a new partition

- `provideStatus()`: represents the serialisation process and is activated when, before a reconfiguration, the worker has to "commit" its partition waiting for a new assignment (if any).

The other two interfaces provided by the user in the worker component are functional interfaces and, in particular, they are the interfaces bound to the multicast ports of the data-parallel skeleton. Thus, their signature is coherent with the ProActive specification:

```
package gridcomp.example.statefulmap.impl;
import gridcomp.map.Task;
import java.util.List;

public interface ServerTestItf1 extends ServerTestItf1 {
    public Task getService(Task t);
}
```

and

```
package gridcomp.example.statefulmap.impl;
import gridcomp.map.Task;
import java.util.List;

public interface ServerTestItf2 extends ServerTestItf2 {
    public Task searchMatch(List<Task> list);
}
```

4.1.4 Running the example

In order to run the example, the user has to follow the steps detailed in Section 2.3 and 2.4.

Running platform In our case, we run the example on a cluster called `pianosau` and the application has been deployed to run on 8 nodes of the cluster called `u12`, `u13`, etc., plus `pianosau` itself as local machine. In order to run the application, from the `GridComp-Examples` directory, the user has to type:

```
~/gridcomp/GridComp-Examples/ > ant remote-deploy-stateful-map
```

for a remote execution, or

```
~/gridcomp/GridComp-Examples/ > ant local-deploy-stateful-map
```

for running the application on the local machine.

Application behaviour The application implements a two-phase protocol:

1. in the first stage, a list of 60 tasks (Integer values) is provided to the getService interface in order to be split among the available workers (5 at starting time)

2. in the second stage, a stream of values are sent to the worker in order to check if they match in any position of the database

The protocol emulates the behaviour of the IBM use-case in which a stream of fingerprints is sent to a distributed database for the recognition process. Each fingerprint (here represented as an Integer value) could match one and only one position of the database distributed among the available processing elements.

Thus, the execution starts with the distribution of the data-set to the workers followed by the first matching queries:

```
~/gridcomp/GridComp-Examples/ > ant local-deploy-stateful-map
remote-deploy-stateful-map:
   ////////////////////////////////////////////////////////
    Runnning on a dataset of 60 tasks.....................
   ////////////////////////////////////////////////////////
    [WORKER-0 on u12] Receiving partition [12-23]
    [WORKER-0 on u13] Receiving partition [24-35]
    [WORKER-0 on u19] Receiving partition [48-59]
    [WORKER-0 on u20] Receiving partition [0-11]
    [WORKER-0 on u14] Receiving partition [36-47]

    [WORKER-0 on u14] Search task 22 in partition [36-47]
    [WORKER-0 on u19] Search task 22 in partition [48-59]
    [WORKER-0 on u13] Search task 22 in partition [24-35]
    [WORKER-0 on u20] Search task 22 in partition [0-11]
    [WORKER-0 on u12] Search task 22 in partition [12-23]

    ---------- Item 22 FOUND ---------
```

In this running example, machines `u13`, `u19`, `u14`, `u12` and `u20` are involved to hosts a worker each (identified by the ID 0), while the manager and the main application component are placed on the user local host by default. Each worker gets a partition of 12 elements (note that these number has been selected for the sake of simplicity) and then start to search in their partition a match with the task they receive. If one of the worker recognise the task, an "ACK" message is sent to the main; if the main receives just an "ACK" message among the ones received by the skeleton on its multicast port, the message `Item ... FOUND` will be printed.

Reconfiguration In the meanwhile, the manager has been started. It accesses the rules file through the JBoss rules engine and detects if one or more rules hold, firing the related action. Thus, during the running the user could have an output like the following one:

```
[MapAutonomicManager] - Detecting information from the enviroment
INFO: Setting QoS information for method searchMatch
[MapAutonomicManager] - Valuating information by fireAllRules()
 ADD_EXECUTOR
```

The first two lines inform the user that the monitoring engine is checking is the observed method `searchMatch` fits the required QoS condition. The fourth line informs about the possible fire of a rule and the fifth row gives the action that will be taken as a consequence of a rule firing. Thus, the execution proceeds with the following lines:

```
    ========== [Start NF-MapAddWorker] ======================

    [MapAddWorker] Binding between multicastServerItf-02 and
                         serverItf-worker-02 completed......
    [MapAddWorker] Binding between multicastServerItf-01 and
                         serverItf-worker-01 completed......

    ============================================================
       Redistribution of dataset from 5 components to 6 components
    ============================================================
    [MapAddWorker] ====== COLLECT partitions from 5 workers ======

    [WORKER-0 on u20] providing the state to the controller...
    [WORKER-0 on u12] providing the state to the controller...
    [WORKER-0 on u13] providing the state to the controller...
    [WORKER-0 on u14] providing the state to the controller...
    [WORKER-0 on u19] providing the state to the controller...
```

```
[MapAddWorker] ===== SPLIT dataset [0-59]

Partition 0: [0-9]
Partition 1: [10-19]
Partition 2: [20-29]
Partition 3: [30-39]
Partition 4: [40-49]
Partition 5: [50-59]

[MapAddWorker] == DISTRIBUTE 60 tasks to 6 workers ==

[WORKER-0 on u20] updating the state from the controller....
[WORKER-0 on u12] updating the state from the controller....
[WORKER-0 on u13] updating the state from the controller....
[WORKER-0 on u14] updating the state from the controller....
[WORKER-0 on u19] updating the state from the controller....
[WORKER-21 on pianosau] updating the state from the controller....

[MapAddWorker] ========= Redistribution COMPLETED ============

================= [End NF-MapAddWorker] ======================
```

The `ADD_EXECUTOR` operation has been fired and in the context of the manager, it means invoking the `MapAddWorker` passive autonomic operation. Such an operation creates a new component, binds it to the two skeleton multicast ports and starts the reconfiguration phase, consisting in:

1. **COLLECTING** and collapsing all the partitions from the pre-existent workers. As mentioned in Section 3 this step requires that the workers provide an autonomic support and implements the `ReconfigSupport` interface in order to correctly serialise the data.
2. **SPLIT** the data with respect to the new available workers
3. **DISTRIBUTE** the new partitions among the new available workers. As mentioned in Section 3, also this step requires that the workers provide an autonomic support and implements the `ReconfigSupport` interface in order to correctly de-serialise the data

Note that, the new component in this run has been placed on the host `pianosau.di.unipi.it` and it has been introduced in the computation process as demonstrated by the successive matching requests that will appear as follows:

```
[WORKER-0 on u19] Search task 16 in partition [40-49]
[WORKER-0 on u14] Search task 16 in partition [30-39]
[WORKER-21 on pianosau] Search task 16 in partition [50-59]
[WORKER-0 on u12] Search task 16 in partition [10-19]
[WORKER-0 on u20] Search task 16 in partition [0-9]
[WORKER-0 on u13] Search task 16 in partition [20-29]

---------- Item 16 FOUND ---------
```

As it can be seen, the new worker `WORKER-21` on `pianosau` is the assigner of the partition spacing from position 50 to 59.

4.2 Stateless data parallel skeleton: the IBM use-case

All the code referenced in the sequel can be found in the package `gridcomp.example.statelessmap.*`, if not differently specified. Moreover, the application implements the contract file (see Section 3) `rules.drl` defined in `gridcomp.example.statelessmap.impl`.

4.2.1 Semantics

The stateless data parallel behavioural skeleton is depicted in Fig. 3 Differently from the stateful

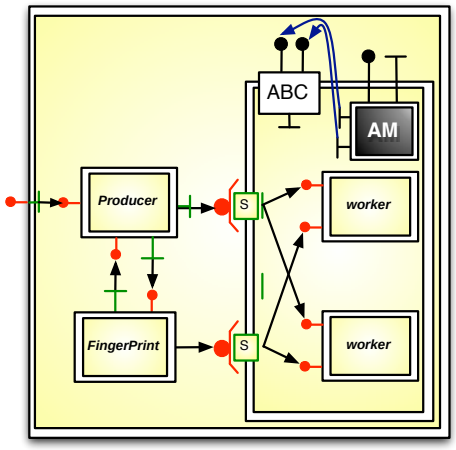

Figure 3: Stateless data-parallel *BeSke* structure

application, here the data parallel skeleton is a component belonging to a composite encapsulating three sub-components: the `Producer` that is in charge of building the data-set to be evaluated by the data parallel skeleton; the `FingerPrint` that is in charge of producing the stream of similar-fingerprints to be recognised by the data-parallel skeleton; the data parallel skeleton itself.
The behavioural skeleton is the same already presented in the previous section and, thus, it focuses on the same semantics requirements; however, in this case, the workers evaluating a partition of the input data-set are stateless unit of computation. From the components point of view, they do not (need to) provide autonomic support. As a consequence, after a reconfiguration, the redistribution of the data set must be functionally managed by explicit user operations. In this example, we will show how to deal with a functional reconfiguration (i.e. with passive autonomicity).

In the following section, we will show how such composite can be used in the user code.

4.2.2 The main application structure

In the stateless example, the main application instantiates the composite by the following piece of code (see Section 4.1.2 for further explanations):

```
Map ctx = new HashMap();
String root;
root = new String("gridcomp.example.statelessmap.adl.testcase");
ctx.put("worker", "gridcomp.example.statelessmap.adl.worker");
ctx.put("rulespath",
  "../GridComp-Examples/src/gridcomp/example/statelessmap/impl/rules.drl");
Component testcase = (Component) f.newComponent(root,ctx);
```

The ADL file representing the component application is defined 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="gridcomp.example.statelessmap.adl.testcase"
   extends="gridcomp.CompositeController"
```

```
   arguments="worker,rulespath">

   <interface signature="gridcomp.example.statelessmap.impl.Producer"
              role="server"
              name="runTestItf"/>

   <!-- first stage -->
   <component name="producer"
              definition="gridcomp.example.statelessmap.adl.producer"/>

   <!-- second stage -->
   <component name="fingerprint"
              definition="gridcomp.example.statelessmap.adl.fingerprint"/>

   <!-- the map -->
   <component name="multicastComposite"
      definition="gridcomp.map.doubleMulticast(${worker},${rulespath})"/>

   <!-- the bindings -->

   <binding client="this.runTestItf"
            server="producer.runProducer"/>
   <binding client="producer.initItf"
            server="multicastComposite.multicastServerItf-01"/>
   <binding client="producer.startfinger"
            server="fingerprint.starter"/>
   <binding client="fingerprint.clientItf"
            server="multicastComposite.multicastServerItf-02"/>
   <binding client="fingerprint.redistribute"
            server="producer.runRedistribute"/>

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

As already mentioned in Section 4.1, the root component extends `gridcomp.Composite-Controller` to provide autonomic features to the component; however, with respect to the stateful example, it encapsulates three sub-components, `producer`, `fingerprint`, `multicastComposite` each representing the three computational units defined in the above section.

Since the data-parallel skeleton exploits the same semantics as the one showed in Section 4.1.2 and that all the observations given above still apply here, we will concentrate our attention on the `producer` and the `fingerprint` component.
The producer component is described by the following user provided ADL 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="gridcomp.example.statelessmap.adl.producer"
   extends="gridcomp.PrimitiveController">

  <interface signature="gridcomp.example.statelessmap.impl.Producer"
             role="server"
             name="runProducer"/>
  <interface signature="gridcomp.map.port.MulticastTestItf1"
             role="client"
             name="initItf"/>
  <interface signature="gridcomp.example.statelessmap.impl.FingerPrint"
             role="client"
```

```
            name="startfinger"/>
  <interface signature="gridcomp.example.statelessmap.impl.Redistribute"
            role="server"
            name="runRedistribute"/>
   <content class="gridcomp.example.statelessmap.impl.ProducerImpl"/>
</definition>
```

The `producer` extends `gridcomp.PrimitiveController`, in order to be defined as a GCM component. Functionally speaking it offers four interfaces, namely:

1. `runProducer` is the hook for the root component to start the application
2. `initItf` is bound to the data parallel skeleton interface devoted to the distribution phase of the generated input data-set
3. `startfinger` is bound to the `fingerprint` component in order to inform it that an initialisation phase or a reconfiguration has just terminated
4. `runRedistribute` is called by to notify the producer that a functional reconfiguration (i.e. a redistribution of the initial state) is required

On the other hand, the `fingerprint` component is described by the following user defined ADL 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="gridcomp.example.statelessmap.adl.fingerprint"
   extends="gridcomp.PrimitiveController">

 <interface signature="gridcomp.map.port.MulticastTestItf2"
            role="client" name="clientItf"/>
 <interface signature="gridcomp.example.statelessmap.impl.FingerPrint"
            role="server" name="starter"/>
 <interface signature="gridcomp.example.statelessmap.impl.Redistribute"
            role="client" name="redistribute"/>

 <content class="gridcomp.example.statelessmap.impl.FingerPrintImpl"/>
</definition>
```

The component exposes three interfaces:

- the server interface `starter` through which the component is notified by the `producer` component that the searching match phase could start
- the client interface `clientItf` through which the component interact with the data parallel skeleton in order to ask for a given matching
- the client interface `redistribute` through which the component notifies to the `producer` that a change in the application structure occurred (maybe the data-parallel skeleton manager removed a worker and its partition of data has been lost) and that a reconfiguration step is needed

The ADL file describing the worker is the same already seen in Section 4.1.2 excepting that it does not provide the `ReconfigSupport` interface, since it is not required that the worker offers autonomic capabilities.

4.2.3 Running the example

The application implements the two-phase (distribution/streaming) protocol already introduced in Section 4.1.4 but:

1. the `producer` is in charge of creating a list of 60 task (Integer values) to be sent to the data parallel skeleton for a parallel computation
2. the skeleton splits the data-set among the available workers
3. the `producer` notifies to the `fingerprint` that the splitting phase has been terminated
4. the `fingerprint` starts pushing matching requests to the skeleton and waits for the set of answers (one per partition): if a request is tagged differently from "ACK" or "NACK", it assumes that a reconfiguration has occurred in the meanwhile and asks the `producer` for a new redistribution. In other words, a worker that has just been added to the processing elements set, signals that he has no "state" to compute on by providing a "fake" tagged response and the `fingerprint` is able to recognise it. Conversely, the fingerprint is able to detect if a worker has been removed from the processing element set as it receives a lower number of responses than expected.
5. the `producer` answers to a redistribution request by reapplying again from point 1

Note that the input has been fixed as a list of 60 tasks just for having an easy and friendly way of presenting the skeleton outcome.

Let us understand how to deal with functional reconfiguration by looking at an execution trace. The QoS contract to which we will refer looks like the following one:

```
package
    gridcomp.manager import gridcomp.manager.beans.*; import
    gridcomp.manager.operations.*; import gridcomp.operation.*; import
    gridcomp.manager.map.impl.PartitionSizeBean;

[methodMonitor="getService"]
rule "CheckHigherBound"
        when
            $arrivalBean : PartitionSizeBean(value >=10)
        then
            $arrivalBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
end
```

The only directive the user provides to the manager is to fire the `ADD_EXECUTOR` autonomic operation (i.e. increase the number of processing elements of the data parallel skeleton) if the value returned by the Java Bean `PartitionSizeBean` (i.e. the size of a partition) is greater than or equal to 10. In other words, the user is asking the manager to add a new processing element to the skeleton if the partition a processing element is working on is greater than 10 tasks (possibly because the load on each processing element is still too high for the user's needs). As a consequence of the increased number of executors and the redistribution process, each partition will decrease in size until the rule condition is no longer verified.
Again, in order to run the example, the user has to follows the steps detailed in Section 2.3 and 2.4.
In our case, we run the example on a cluster called `pianosau` and the application has been deployed to run on 8 nodes of the cluster called `u2-u10`, plus `pianosau` itself as local machine. In order to run the application from the `GridComp-Examples` directory, the user has to type:

```
~/gridcomp/GridComp-Examples/ > ant remote-deploy-stateless-map
```

for a remote execution, or

```
~/gridcomp/GridComp-Examples/ > ant local-deploy-stateless-map
```

for running the application on the local machine.

Thus, the execution starts with the distribution of the data-set to the workers followed by the first matching queries:

```
Buildfile: build.xml

remote-deploy-stateless-map:
     ////////////////////////////////////////////////////////
      Runnning on a dataset of 60 tasks.....................
     ////////////////////////////////////////////////////////
     [Worker 0 on u14] Receiving partition [36-47]
     [Worker 0 on u12] Receiving partition [12-23]
     [Worker 0 on u19] Receiving partition [48-59]
     [Worker 0 on u20] Receiving partition [0-11]
     [Worker 0 on u13] Receiving partition [24-35]

     [ProducerImpl] End of data partitioning!

     ------- [FingerPrint] Ask for recognition of item 55 -------
     [WORKER-0 on u20] Receiving partition [0-11] search for item 55
     [WORKER-0 on u19] Receiving partition [48-59] search for item 55
     [WORKER-0 on u14] Receiving partition [36-47] search for item 55
     [WORKER-0 on u13] Receiving partition [24-35] search for item 55
     [WORKER-0 on u12] Receiving partition [12-23] search for item 55

     [FingerPrint] Item FOUND!
```

Eventually, the manager evaluates the conditions defined in the QoS contract, meaning that the Java Bean `PartitionSizeBean` is evaluated and if it returns a value that satisfies the QoS condition, the `ADD_EXECUTOR` operation is fired, causing the execution of the autonomic operation `MapAddExecutor`, as can be seen in the following lines:

```
    [MapAutonomicManager] - Detecting information from the environment
    Setting QoS information for method getService
    [MapAutonomicManager] - Valuating information by fireAllRules()
     ADD_EXECUTOR
```

What follows is related to the execution of the autonomic operation

```
     ================== [Start NF-MapAddWorker] ====================

     [MapAddWorker] Binding between multicastServerItf-02
                           and serverItf-worker-02 completed......
     [MapAddWorker] Binding between multicastServerItf-01
                           and serverItf-worker-01 completed......

     [MapAddWorker] Stateless computation running, skip redistribution

     ================== [End NF-MapAddWorker] =======================
```

The new worker is instantiated and bound to the skeleton multicast interfaces; since the new worker does not support autonomicity features (or, equally, the pre-existent workers don't), the manager can't define a redistribution of the partition, thus the non-functional operation terminates and the functional computation proceeds.

```
     ------- [FingerPrint] Ask for recognition of item 26 -------
     [WORKER-0 on u14] Receiving partition [36-47] search for item 26
     [WORKER-0 on u12] Receiving partition [12-23] search for item 26
```

```
[Worker 21 on pianosau] no tasks to compute, yet
[WORKER-0 on u20] Receiving partition [0-11] search for item 26
[WORKER-0 on u13] Receiving partition [24-35] search for item 26

[FingerPrint] Item FOUND!
```

As it can be seen by the output trace, the new worker (allocated on `pianosau`) is involved in the computation but he is not able to answer to the matching requests because it does not hold any partition. As the `fingerprint` component detects the problem, it ask the `producer` component for a redistribution step:

```
[FingerPrint] Redistribution needed.....

[Worker 21 on pianosau] Receiving partition [50-59]
[Worker 0 on u14] Receiving partition [30-39]
[Worker 0 on u12] Receiving partition [10-19]
[Worker 0 on u13] Receiving partition [20-29]
[Worker 0 on u19] Receiving partition [40-49]
[Worker 0 on u20] Receiving partition [0-9]
[ProducerImpl] End of DB RE-partitioning!
```

At this point, the computation can proceed involving 6 components, holding a given partition each.

```
------- [FingerPrint] Ask for recognition of item 26 -------
[WORKER-21 on pianosau] Receiving partition [50-59] search for item 26
[WORKER-0 on u19] Receiving partition [48-59] search for item 26
[WORKER-0 on u20] Receiving partition [0-9] search for item 26
[WORKER-0 on u12] Receiving partition [10-19] search for item 26
[WORKER-0 on u13] Receiving partition [20-29] search for item 26
[WORKER-0 on u19] Receiving partition [40-49] search for item 26
[WORKER-0 on u14] Receiving partition [30-39] search for item 26

[FingerPrint] Item FOUND!
```

4.2.4 Guidelines for using the stateless data parallel skeleton

The steps a user must follow in order to successfully use the stateless data parallel skeleton can be summarised as follows:

1. provide the root component description as an extension of the `gridcomp.CompositeController` (and any other sub-component he/she wants to use);
2. and provide the worker component description as an extension of the `gridcomp.PrimitiveController` providing
3. include in his application definition the `gridcomp.map.doubleMulticast` version of the skeleton, caring of using its ports in the right way, as described in Section 4.1.2
4. provide the needed parameters (the ADL file of the worker and the path to the QoS contract file) to the skeleton at running time
5. treating at functional level the state updates of the workers in case of reconfiguration

If a worker is removed (all but 1 workers can be removed), the user will find a similar behaviour, consisting in a collecting, splitting and redistribution phase among the remaining workers.

4.2.5 Guidelines for using the stateful data parallel skeleton

The steps a user must follow in order to successfully use the stateful data parallel skeleton can be summarised as follows:

1. provide the root component description as an extension of the `gridcomp.CompositeController` (and any other sub-component he/she wants to use);
2. and provide the worker component description as an extension of the `gridcomp.PrimitiveController` providing, among the others, the `gridcomp.map.controller.operation.ReconfigSupport` interface in order to instruct the manager about how serialise/de-serialise the worker state
3. include in his application definition the `gridcomp.map.doubleMulticast` version of the skeleton, caring of using its ports in the right way, as described in Section 4.1.2
4. provide the needed parameters (the ADL file of the worker and the path to the QoS contract file) to the skeleton at running time

5 Implementing a farm behavioural skeleton

5.1 Farm skeleton at passive level: the Mandelbrot set

In this section we will show how an application exploiting a farm parallelism pattern can be programmed into the framework with GCM behavioural skeletons.

5.1.1 Semantics

We will take the Mandelbrot application in consideration as a model of programming these patterns of computation: it performs calculations to display dynamically the Mandelbrot fractal image. We will show how passive autonomicity can be expressed in the context of a real application.

All the code referenced in the sequel belongs to the package `gridcomp.example.passive.mandelbrot`, if not differently specified.

5.1.2 The main application structure

The application structure is depicted in Fig. 4

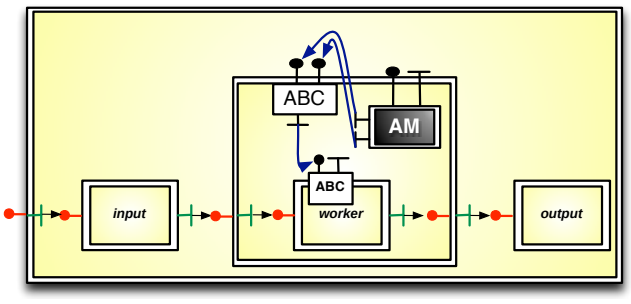

Figure 4: Mandelbrot application composite.

The ADL file representing the root component has the following content:

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

```
  <definition name = "gridcomp.example.passive.mandelbrot.adl.root"
              extends = "gridcomp.CompositeController">

  <component name="input"
             definition = "gridcomp.example.passive.mandelbrot.adl.input"/>
  <component name="farm"
             definition = "gridcomp.example.passive.mandelbrot.adl.farm"/>
  <component name="output"
             definition = "gridcomp.example.passive.mandelbrot.adl.output"/>

  <binding client = "farm.collector-client"
           server = "output.collector-server"/>
  <binding client = "input.lineserver-client"
           server = "farm.lineserver-server"/>
  <virtual-node name="master-node" cardinality="single"/>
</definition>
```

This application exhibits a passive autonomicity and the root component encapsulates an `input` component producing the stream of task on which evaluate the Mandelbrot function, the `farm` component and the `output` component that is in charge of providing a graphical output to the computation.

The input component The `input` component is defined as follows in the `input.fractal` file

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

<definition name = "gridcomp.example.passive.mandelbrot.adl.input"
   extends = "gridcomp.PrimitiveController">

  <interface signature = "gridcomp.example.passive.mandelbrot.LineServer"
             role = "client" name = "lineserver-client"/>

  <content class="gridcomp.example.passive.mandelbrot.Input"/>

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

The worker component The ADL of the worker component provided by the user is defined as follows:

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

  <definition name="gridcomp.example.passive.mandelbrot.adl.lineserver"
              extends="gridcomp.PrimitiveController">

  <interface signature="gridcomp.example.passive.mandelbrot.LineCollector"
             role="client" contingency="optional" name="collector-client"/>
  <interface signature="gridcomp.example.passive.mandelbrot.LineServer"
             role="server" name="lineserver-server"/>
  <content class="gridcomp.example.passive.mandelbrot.LineServerImpl"/>
  <virtual-node name="slave-1" cardinality="single"/>
</definition>
```

It extends the `gridcomp.PrimitiveController` in order to inherit all the GCM controller and it defines on which virtual node the worker has to be placed at deployment time.
For each tasks received, the `lineserver` component performs some iterations and returns to the `output` component the result computed. The application ends when all the points of the Mandelbrot set have been computed.

Each worker is coupled with an ADL descriptor needed to express how to map the application onto the target architecture. In the example above, the ADL descriptor is represented by the `slave-1.fractal` file, that appears 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="gridcomp.example.passive.mandelbrot.adl.slaves.slave-1"
  extends="gridcomp.example.passive.mandelbrot.adl.lineserver">

 <virtual-node name="slave-node-1" cardinality="single"/>
</definition>
```

As it can be seen, the file simply maps a specific worker instance on a given virtual node of the ProActive platform.

The farm skeleton As we could expect, the `farm` component has the following definition:

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

  <definition name="gridcomp.example.passive.mandelbrot.adl.farm"
              extends="gridcomp.controller.MonitorBalanceFarmController">

  <interface signature="gridcomp.example.passive.mandelbrot.LineCollector"
             role="client" contingency="optional" name="collector-client"/>
  <interface signature="gridcomp.example.passive.mandelbrot.LineServer"
             role="server" name="lineserver-server"/>

  <component name = "server"
             definition = "gridcomp.example.passive.mandelbrot.adl.lineserver"/>

  <binding server = "this.collector-client"
           client = "server.collector-client"/>
  <binding client = "this.lineserver-server"
           server = "server.lineserver-server"/>

  <virtual-node name="master-node" cardinality="single"/>

  <beske xmlFile="src/gridcomp/example/passive/mandelbrot/farm.properties"/>
</definition>
```

The component ADL definition is enriched with the new GCM element, *BeSke*. Such element points to an existing file which configures this particular instance of the behavioural skeleton. In particular, it specifies the directory where the ADL deployment descriptors describing the mappings between workers and virtual nodes and the pathname of the XML deployment file to be used when creating a new worker instance. This is the content of the *BeSke* file related to the π example

```
adl.dir = gridcomp/example/passive/pi/adl/slaves
xml.file = ./distrib-deployment.xml
```

5.1.3 Running the example

Since the application exploits a passive autonomicity, users do not submit any QoS contract to the application. However, the application provides a GUI allowing to drive the autonomic behaviour through a control panel offering a set of buttons for increasing/decreasing the number of workers, query some monitoring information and activate the load balancing among workers after a reconfiguration (see Fig. 5). The user can take such decisions by monitoring the configu-

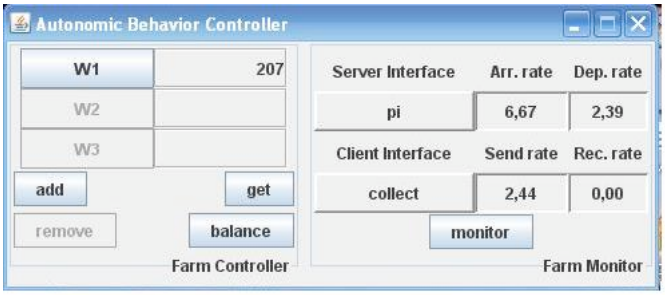

Figure 5: Control panel driving the passive autonomicity of the Mandelbrot application.

ration and the performance of the application through a monitoring window on his desktop (see Fig.7) showing how, as the number of workers changes in time, the throughput of the application increases/decreases. A picture of the graphical output of the application is shown in Fig. 6,

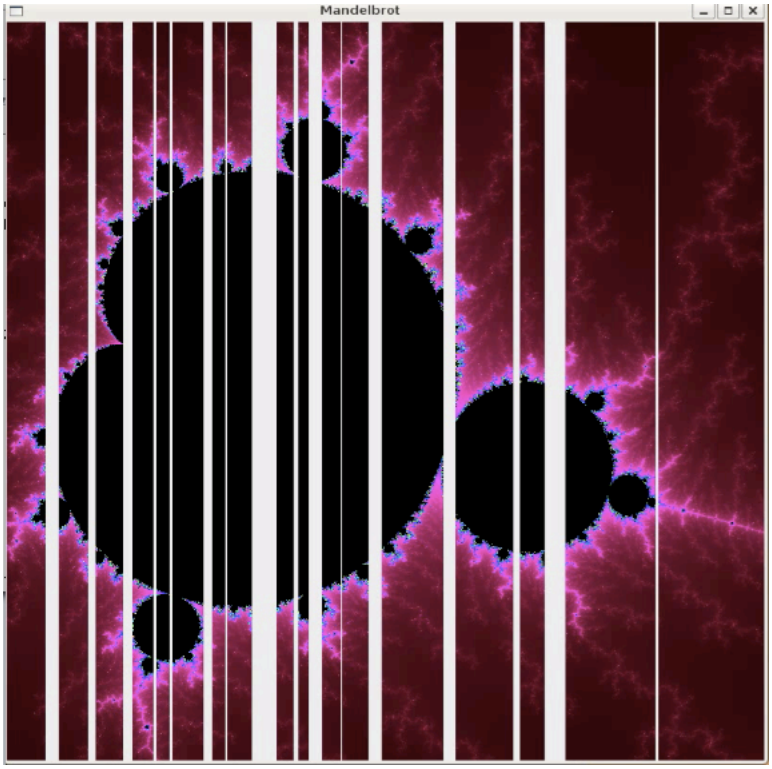

Figure 6: Mandelbrot application output.

where an animated window incrementally shows the fractal image derived from the calculation. The images has been captured from an execution running on a cluster called `pianosau` and the

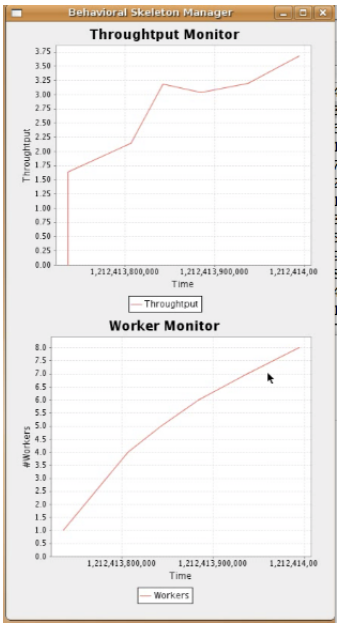

Figure 7: The monitoring window of the Mandelbrot application shows how, as the number of workers change in time, the throughput of the application increases/decreases.

application has been deployed to run on 8 nodes of the cluster called `u2-u10`, plus `pianosau` itself as local machine. In order to run the application, from the `GridComp-Examples` directory, the user has to type:

```
~/gridcomp/GridComp-Examples/ > ant remote-deploy-passive-mandelbrot
```

for a remote execution, or

```
~/gridcomp/GridComp-Examples/ > ant local-deploy-passive-mandelbrot
```

for running the application on the local machine.

5.1.4 Guidelines for using the farm skeleton at passive level

The steps a user must follow in order to successfully use the farm skeleton can be summarised as follows:

1. provide the root component description as an extension of the `gridcomp.CompositeController` (and any other sub-component he/she wants to use);
2. provide the worker component description as an extension of the `gridcomp.PrimitiveController`; for each instanciable worker, a descriptor for the deployment phase must be provided.
3. include in his application definition of the farm skeleton as an extension of the `gridcomp.controller.MonitorBalanceFarmController` as showed in the example above
4. providing the `farm.properties` file and set of ADL files describing the map between each instanciable worker and the virtual nodes

5.2 Farm skeleton at active level: the DiCom use-case

In this section we will show an example of active autonomicity involving the farm behavioural skeleton, i.e. an application in which the non-functional strategies related to the reconfiguration of the composite are taken by a manager belonging to the skeleton itself.

5.2.1 Semantics

In order to illustrate how active autonomicity works, we will show how the so called DiCom use-case has been implemented by exploiting the farm behavioural skeleton provided by the autonomic framework. The use-case is related to an application for the recognition of cancer areas to be detected on a stream of images representing mammary x-rays.

The use-case will show how active autonomicity is provided by the farm skeleton and we will highlight the differences with respect to the passive case.

5.2.2 The main application structure

The root component is described by the following ADL file that takes the QoS contract as parameter, as already shown in Section 4.1:

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

  <definition name="gridcomp.example.active.dicom.adl.root"
              extends="gridcomp.CompositeController"
              arguments="contractFile">

  <interface signature="gridcomp.example.active.dicom.Master"
             role="server" name="do"/>
  <component name="client"
             definition="gridcomp.example.active.dicom.adl.master"/>
  <component name="farm"
             definition="gridcomp.example.active.dicom.adl.farm($(contractFile))"/>
  <component name="output"
             definition="gridcomp.example.active.dicom.adl.output"/>

   <binding client="this.do"
           server="client.do"/>
  <binding client="client.work"
           server="farm.work"/>
  <binding client="farm.send"
           server="output.send"/>
  <virtual-node name="master-node" cardinality="single"/>
</definition>
```

The component encapsulates three components: the `client` component producing the stream of images, the `farm` component evaluating them and the `output` component that is in charge of providing a graphical output of the elaboration.

Although the farm is not in the scope of the user responsibility, we will show how the farm component is structured in order to ease the explanation. The farm component has the following description

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

  <definition name="gridcomp.example.active.dicom.adl.farm"
              extends="gridcomp.controller.MonitorBalanceFarmController">

  <interface name="work" role = "server"
             signature = "gridcomp.example.active.dicom.Worker" />
  <interface name="send" role = "client"
             signature = "gridcomp.example.active.dicom.Collector" />
```

```
  <component name="worker"
             definition = "gridcomp.example.active.dicom.adl.worker"/>

  <component name="manager"
             definition="gridcomp.manager.farm.adl.AutonomicManager(${contractFile})" />

 <binding client="manager.client-autonomic-controller"
          server = "this.autonomic-controller"/>
  <binding client="manager.client-beske-attribute-controller"
          server = "this.beske-attribute-controller"/>

  <binding client="this.work"   server = "worker.work"/>
  <binding client="worker.send" server = "this.send"/>

  <virtual-node name="master-node" cardinality="single"/>

   <beske xmlFile="src/gridcomp/example/active/dicom/farm.properties"/>
</definition>
```

The farm is parametric with respect to the QoS contract (the JBoss rule file) that will be given to the `AutonomicManager` as input file. The manager is bound to the GCM controllers of the farm and the skeleton is provided with its own file of properties defined in `farm.properties`, specifying the ADL file describing the worker and the path to the XML deployment file. The

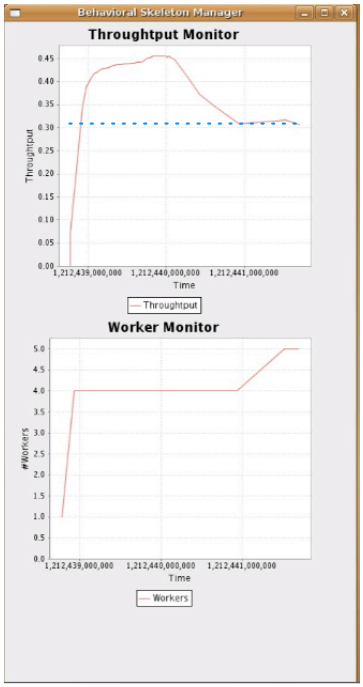

Figure 8: A perspective on the autonomic behaviour of the DiCom use-case. The throughput monitor window shows the contractually required throughput (dashed line) and the actual throughput of the farm along the execution. The worker monitor shows the how the number of worker vary (increase in this case) to reflect throughput variation (drop). In the experiment the throughput drop is caused by an additional load happening in the platforms running farm workers (here artificially induced by an external application).

contract file has the following structure:

```
package gridcomp.computeTaskMonitor
import gridcomp.manager.beans.*;
import gridcomp.manager.operations.*;
import gridcomp.operation.*;

[methodMonitor="computeTask"]
rule "CheckRate"
        when
            $arrivalBean : DepartureRateBean(value < 1.2)
        then
            $arrivalBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
            $arrivalBean.fireOperation(ManagerOperation.BALANCE_LOAD);
end

rule "StartingParDegree"
when
$parDegree: NumWorkerBean(value < 4)
then
  $parDegree.fireOperation(ManagerOperation.ADD_EXECUTOR);
                  $parDegree.fireOperation(ManagerOperation.ADD_EXECUTOR);
                  $parDegree.fireOperation(ManagerOperation.ADD_EXECUTOR);
                  $parDegree.fireOperation(ManagerOperation.BALANCE_LOAD);
end
```

The observed method, `computeTask` belongs to the worker's interface:

```
package gridcomp.example.active.dicom;
public interface Worker {
public final static String ITF_NAME = "work";
public void computeTask(byte[] data, int idx);
}
```

Two rules have been defined on the manager, implementing two different events to check and two different actions to take. Both the Java Beans involved evaluate a given condition by querying the autonomic behaviour controller (ABC) provided with the farm. The first rule assesses that if the departure rate of the tasks (given by the `DepartureRateBean` Java Bean) is lower than a specified threshold, then two actions must be taken at passive level: the instantiation of a new worker followed by the load balancing of the tasks between the existing workers.
The second rule assesses that if the number of workers (given by the `NumWorkerBean` Java Bean is lower than a given threshold, two workers have to be added to the farm and their tasks loads must be balanced.

As a consequence of this QoS contract, the manager drives the execution so that the number of workers encapsulated by the farm is at least 4 and, however, such number is kept sufficiently high to guarantee a given performance expressed in terms of departure rate.

5.2.3 Running the example

The execution of the DiCom use-case is graphically supported by a GUI showing two windows: on the first window (Fig.9) , the result of the execution can be appreciated as a sequence x-ray images, whose scroll speeds as the overall performance of the application improves.

On the second window (Fig. 8), two monitoring charts about throughput and number of workers involved, give the possibility to the user to follow the application behaviour under an autonomic perspective. In particular, the animated window shows how, as the number of workers changes in time, the throughput of the application increases/decreases.
The pictures have been captured from an execution running on a cluster called `pianosau` and the application has been deployed to run on 8 nodes of the cluster called `u2-u10`, plus `pianosau`

Figure 9: The sequence of x-ray images produced by the application.

itself as local machine. In order to run the application, from the `GridComp-Examples` directory, the user has to type:

```
~/gridcomp/GridComp-Examples/ > ant remote-deploy-dicom
```

for a remote execution, or

```
~/gridcomp/GridComp-Examples/ > ant local-deploy-dicom
```

for running the application on the local machine.

5.2.4 Guidelines for using the farm skeleton at active level

The steps a user must follow in order to successfully use the farm skeleton are the same already listed in 5.2.4. However, in this case the user must take care of

1. selecting a version of the farm that includes the manager as an inner component
2. provide the JBoss rules file to address the monitoring issues and pass it to the root component as a parameter at running time.

6 Conclusion and work in progress

We presented a set of use-cases showing how the autonomic features within the GCM implementation given on top of ProActive 3.9 can be exploited in a parallel programming application. As the traditional skeleton paradigm, the basic idea is to provide the user with a set of behavioural skeleton, i.e. components whose non-functional behaviour is pre-defined, while the functional behaviour is parametrically specified by the user.

Since the non-functional behaviour of such component is pre-defined, we are able to provide a manager driving the behaviour with respect to the user's needs specified through a QoS contract at running time.

The document presents two use-cases related to the data parallel skeleton and two use-cases related to the farm skeleton, in which passive and active autonomicity has been exploited.

We are currently working on allowing the user (or the framework programmer) to customise its behavioural skeletons. In particular, in the next future he/she will be able to:

- extending or overwriting the manager of the provided skeleton in order to add/change manager capabilities while keeping the skeleton structure and behaviour;
- adding new autonomic operation to both the active and the passive level in order to increase the adaptive power of pre-existent or new skeletons;
- adding new controllers in order to wide the spectrum of autonomic operations at passive level as well as monitoring capabilities.

Besides the new Priority Controller introduced in ProActive3.9 we introduced some new controllers for components to implement the effector and the sensor of the behavioural skeletons (as an example the monitor controller). These controllers strictly depend on the Proactive implementation of components, and they are generic enough to be reusable by other *BeSke* programmers (exploiting GCM/PROACTIVE implementation). In the future, we plan to substitute such controllers with the ones provided by our project partners in the next months.

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Peter Kilpatrick, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons for component autonomic management on grids. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, Greece, June 2007.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In Didier El Baz, Julien Bourgeois, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, February 2008. IEEE.

[3] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.NFCF.01 – Non functional component subsystem architectural design*, June 2007. `http://gridcomp.ercim.org/images/stories/Deliverables/d.nfcf.01-final.pdf`.

[4] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, February 2007. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[5] JBoss rules home page. `http://www.jboss.com/products/rules`, 2008.

[6] ObjectWeb Consortium. *The Fractal Component Model, Technical Specification*, 2003.

[7] ProActive home page, 2006. `http://www-sop.inria.fr/oasis/proactive/`.

[8] Thomas Weigold, Peter Buhler, Jeyarajan Thiyagalingam, Artie Basukoski, and Vladimir Getov. Advanced grid programming with components: A biometric identification case study. In *Proc. of the 32nd Intl. Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, 2008. IEEE. To appear.