Chapter 6

Structured parallel computing lab

6.1 Implementation programming environments

6.1.1 POSIX TCP/IP

6.1.2 ProActive

Environment setup

In order to use ProActive, you must download the library from the ProActive web site at `http://proactive.inria.fr`. Follow the instructions on the web site and eventually you'll get a zip or tgz file named `ProActive-`*versionNo*`.tgz` This is the only thing you need to run code with the library. The following steps are needed to complete environment setup (we'll refer here to ProActive version 4.0.2, the one available when these notes have been written):

1. unpack the file. You'll get a `ProActive-`*versionNo* directory. This will be the `PROACTIVE_HOME` directory.

2. add to the `CLASSPATH` all the jar files in `PROACTIVE_HOME/dist/lib`, including the `ProActive.jar`. Actually, to run the base examples, only the jar files `ProActive.jar`, `javassist.jar`, `log4j.jar`, `xercesimpl.jar` and `bouncycastle.jar` are needed, plus the `fractal.jar` which is needed to develop component based programs.

3. if you use Eclispe, add the same jar file to the `Library` entry in the project `Properties`

4. when running programs, remember to pass the JVM at least the following arguments:

- `-Dfractal.provider=org.objectweb.proactive.core.component.Fractive`, which is used to define the Fractal base environment
- `-Djava.security.policy= ...` pointing yo a file hosting the permissions relative to the security policies to be adopted when executing programs. Typically, if you are the only user of the environment, you would like to have a `-Djava.security.policy=file.policy` where the `file.policy` hosts the lines:

  ```
  grant {
    permission java.security.AllPermission;
  };
  ```

  that *de facto* nullify all the security controls.
- `-Dproactive.home=...` pointing to the `PROACTIVE_HOME` directory

Sample code: using a single, simple component

A primitive component only providing server ports is defined providing

1. a Java interface, with the interface exposed to the component framework by the component
2. a Java class implementing the component itself,
3. a `.fractal` component descriptor, stating all the properties of the component. In particular, the file will specify the component ports (client and server) as well as the implementation classes used to implement the component itself.

Let us assume we want to implement a component providing a very simple service: computing the successor of an Integer passed as argument of the service call.

The interface exposed by the component will be (`SimpleComponentInterface.java`):

Listing 6.1: SimpleComponentInterface.java

```
1 package adles2;
2
3 public interface SimpleComponentInterface {
4
5   public Integer doWork(Integer task);
6 }
```

The implementation of the component is given by the correspondent `SimpleComponent.java` code:

Listing 6.2: SimpleComponent.java

```
1 package adles2;
2
3 public class SimpleComponent implements SimpleComponentInterface {
4
5   public Integer doWork(Integer task) {
6     int iv = task.intValue();
7     System.err.println("--- doWork computing "+iv+" got task "+task
          +":"+task.getClass().getName());
8     Integer r = new Integer(++iv);
9     return r;
10   }
11
12 }
```

Eventually the component descriptor can be given as follows:

Listing 6.3: SimpleComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
     EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
      proactive.dtd">
4
5 <definition name="adles1.SimpleComponent">
6
7   <interface
8     name="simple-comp-interface"
9     signature="adles1.SimpleComponentInterface"
10     role="server" />
11
12   <content
13     class="adles1.SimpleComponent" />
14
15   <controller
16     desc="primitive" />
17
18 </definition>
```

This is the most important part, probably. The descriptor shown here is the simplest possible. Lines 1 to 3 just define the XML schemas and DTDs to be used. Line 5 gives a name to the component. As this component is part of the package `adles1` the name is a fully qualified Java class name, although it could have been anything else. This name is the one to be used when using this component in other component assemblies. Lines 7 to 10 define the interface exposed by the component. The interface is given a name, which will be used when instantiating the component to retrieve its ports, a signature, pointing to the Java class defining the interface, and a role. The role may be `client` or `server`, for use and provide ports, respectively. Lines 12 to 13 are used to provide an implementation to the component. This is the fully qualified Java class name hosting the implementation (code in Listing 6.2, in our case). Eventually, line 18 denotes the fact the component is primitive, that is it has no subcomponents.

These 3 files altogether define our first Fractal/GCM component in ProActive. In order to use the component, we can pass the ADL file to a proper Factory method, as in the following sample usage code:

Listing 6.4: Main.java

```
1 package adles1;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 import org.objectweb.fractal.adl.Factory;
7 import org.objectweb.fractal.api.Component;
8 import org.objectweb.fractal.util.Fractal;
9
10 public class Main {
11
12   public static void main(String [] args) {
13
14     try {
15       // get factory to instantiate the component from ADL
16       Factory f = org.objectweb.proactive.core.component.adl.
             FactoryFactory.getFactory();
17       // the has table is used to set up the context
18           Map<String, Object> context = new HashMap<String, Object
                 >();
19
20           Component simpleComponent = (Component) f.newComponent("
                 adles1.SimpleComponent", context);
21           System.out.println(":: Component created ...");
22
23           Fractal.getLifeCycleController(simpleComponent).startFc();
24       System.out.println(":: Component started ...");
25
26
27       SimpleComponentInterface sc = ((SimpleComponentInterface)
             simpleComponent.getFcInterface("simple-comp-interface"));
28       Object n = sc.doWork(new Integer(5));
29       System.out.println(":: Object coming back of type : "+n.
             getClass().getName());
30
31       System.out.println(":: Computed sc.doWork(5)="+n.toString());
32
33
34           Fractal.getLifeCycleController(simpleComponent).stopFc();
35       System.out.println(":: Component stopped ...");
36
37       System.exit(0);
38
39     } catch (Exception e) {
40       e.printStackTrace();
41       System.exit(0);
42     }
43
44   }
45
46 }
```

A Factory is taken from the component framework at lines 16. It is then used to get a Component at lines 20. The second parameter is for setting up context in the environment. The first one is the ADL file (Listing 6.3 above). It has to be located in the classpath where the corresponding Java class will be eventually looked for. This means, that if you use to have a src and a bin directory for the Java files and for the compiled class files, the .fractal file has to be placed in the bin directory, not in the src one. Be careful, when renaming packages under Eclipse, that renaming moves only the .class files and therefore if you rename a package with .fractal files inside, these files will be lost after renaming. Once we have got the component, we should start it. We'll therefore get the lifecycle controller of the component and apply the startFC method (line 23). Only after executing the startFc method of the controller the component will be available to serve doWork requests.

Lines 27 to 31 show sample component usage. In order to be able to invoke the component server port, we need to retrieve the component interface. This is done in line 27. The name of the interface here is the name specified in the SimpleComponent.fractal component descriptor. At this point we can invoke the server port (line 28) and get a result out of the component service. Line 34 stop the component and the program terminates at line 37.

The output of the program, when run through ProActive, is the following:

Listing 6.5: sample output

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.adles1._StubSimpleComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 :: Component created ...
6 :: Component started ...
7 Generating class : pa.stub.java.lang._StubObject
8 :: Object coming back of type : pa.stub.java.lang._StubObject
9 --- doWork computing 5 got task 5:java.lang.Integer
10 :: Computed sc.doWork(5)=6
11 :: Component stopped ...
```

Lines starting with :: or --- are from the sample code, the other ones are the messages from the ProActive code.

Sample usage: composite component

Now we consider a slightly different example. We use the SimpleComponent discussed before with another component, introduced here, to show how a component assembly can be built. The second component introduced will use the first component. Therefore it will implement *use* (client) ports as well. This, in turn, will introduce the necessity to implement a BindingController within the component.

Assume the first component is the SimpleComponent defined above. We want to implement another component that multiplies by two the value obatined

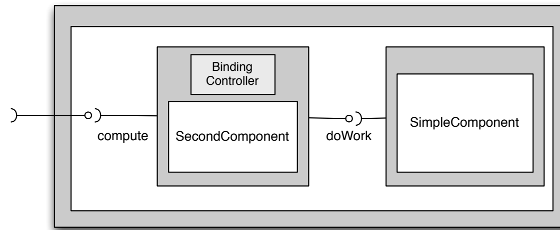

Figure 6.1: Simple component assembly

passing the parmeter to the first component. The assembly we want to implement is therefore the one in Fig.6.1.

Let us define the second component. It will have a use and a provide interface. The use interface should be eventually bind to the provide interface of the other component, while the provide interface will be eventually promoted to be accessible outside the assembly component. The interface of the `SecondComponent` is defined as follows:

Listing 6.6: SecondComponentInterface.java

```
1 package adles2;
2
3 public interface SecondComponentInterface {
4
5   public Integer compute(Integer task);
6 }
```

It is worth pointing out that the interface *does not expose* the use ports, but only the provide ones. Use ports will be exposed in the component descriptor:

Listing 6.7: SecondComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="adles2.SecondComponent">
6
7   <interface
8     name="second-comp-interface"
9     signature="adles2.SecondComponentInterface"
10    role="server" />
11
12  <interface
13    name="simple-comp-interface"
14    signature="adles2.SimpleComponentInterface"
15    role="client" />
16
17  <content
```

```
18     class="adles2.SecondComponent" />
19
20 </definition>
```

In this descriptor, the provide interface is described as in the single component example above, and the use interface is described using the same tags but assigning it a `client` role.

In order to implement `SecondComponent` we must implement also the `BindingController` interface, however. This is because the use interface in the descriptor will be bound to the provide interface of the other component during the component assembly. This binding will be actually performed using the `BindingController`.

Therefore a very simple implementation of the `SecondComponent` can be the following one:

Listing 6.8: SecondComponent.java

```
1  package adles2;
2
3  import org.objectweb.fractal.api.NoSuchInterfaceException;
4  import org.objectweb.fractal.api.control.BindingController;
5  import org.objectweb.fractal.api.control.IllegalBindingException;
6  import org.objectweb.fractal.api.control.IllegalLifeCycleException;
7
8  public class SecondComponent implements SecondComponentInterface,
       BindingController {
9
10   SimpleComponentInterface otherComponent;
11
12   public SecondComponent() {
13     // empty
14     return;
15   }
16
17   @Override
18   public Integer compute(Integer task) {
19     // call other component
20     Integer temp = otherComponent.doWork(task);
21     System.err.println("--- called SimpleComponent, got "+temp+"
         type "+temp.getClass().getName());
22     // post process result
23     int i = temp.intValue();
24     // return new result
25     return new Integer(i*2);
26   }
27
28   @Override
29   public void bindFc(String arg0, Object arg1)
30       throws NoSuchInterfaceException, IllegalBindingException,
31       IllegalLifeCycleException {
32
33     if(arg0.equals("simple-comp-interface")) {
34       otherComponent = (SimpleComponentInterface) arg1;
35     }
36   }
```

```
37
38    @Override
39    public String[] listFc() {
40      String [] itf = new String[1];
41      itf[0] = "simple-comp-interface";
42      return itf;
43    }
44
45    @Override
46    public Object lookupFc(String arg0) throws
          NoSuchInterfaceException {
47      if(arg0.equals("simple-comp-interface")) {
48        return otherComponent;
49      }
50      return null;
51    }
52
53    @Override
54    public void unbindFc(String arg0) throws NoSuchInterfaceException,
55        IllegalBindingException, IllegalLifeCycleException {
56
57      if(arg0.equals("simple-comp-interface")) {
58        otherComponent = null;
59      }
60  }
61
62
63
64
65  }
```

Line 10 defines the private instance variable that will eventually host the reference to the component providing the server interface eventually bind to this component use interface. The binding is implemented through the `bindFc` method of the `BindingController`. The name of the interface to be bound here is the one appearing into the `.fractal` component descriptor. The `compute` method, the one exposed as provide port in the descriptor, gets an `Integer` parameter, passes it to the other component (line 20) and eventually returns the number received back multiplied by 2 (line 25). It is probably convenient to point out that the `otherComponent` has type `SimpleComponentInterface` rather than `SimpleComponent`, as the component only needs to know the exported port to use other components. The `unbindFc`, `listFc` and `lookupFc` methods are also required by the `BindingControllerInterface`.

Now that we have both components, it is time to consider the composite component assembly. This is defined through the following `Composite.fractal` descriptor

Listing 6.9: Composite.fractal

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

```
4
5 <definition name="adles2.Composite">
6
7   <interface
8     name="outer-comp-interface"
9     signature="adles2.SecondComponentInterface"
10    role="server" />
11
12  <component
13    name="second-component"
14    definition="adles2.SecondComponent" />
15
16  <component
17    name="first-component"
18    definition="adles2.SimpleComponent" />
19
20  <binding
21    client="second-component.simple-comp-interface"
22    server="first-component.simple-comp-interface" />
23
24  <binding
25    client="this.outer-comp-interface"
26    server="second-component.second-comp-interface" />
27
28 </definition>
```

The interface defined here is the interface of the composite component. It does not use anything else, so it only sports server interfaces (just one, in this case). The two components used in the assembly are denoted using *references*. Instead, they could have been denoted using a `<component> ... </component>` tag. However, this way of denoting sub-components allow full re usage of the original sub-components descriptor. The extra tags in the assembly provide bindings among the different ports involved. The first `binding` actually connects use port of the `SecondComponent` to the corresponding provide port of the `SimpleComponent`. The second one, promotes the provides port of the `SecondComponent` as a provide port of the composite component.

Let's have a look at some code using this composite.

Listing 6.10: Main.java

```
1 package adles2;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 import org.objectweb.fractal.adl.Factory;
7 import org.objectweb.fractal.api.Component;
8 import org.objectweb.fractal.util.Fractal;
9
10 public class Main {
11
12   public static void main(String [] args) {
13
14     try {
15       // get factory to instantiate the component from ADL
```

```
16        Factory f = org.objectweb.proactive.core.component.adl.
              FactoryFactory.getFactory();
17        // the has table is used to set up the context
18            Map<String, Object> context = new HashMap<String, Object
                  >();
19
20            Component composite = (Component) f.newComponent("adles2.
                  Composite", context);
21            System.out.println(":: Composite Component created ...");
22
23            Fractal.getLifeCycleController(composite).startFc();
24        System.out.println(":: Component started ...");
25
26
27        SecondComponentInterface sc = ((SecondComponentInterface)
              composite.getFcInterface("outer-comp-interface"));
28        Object n = sc.compute(new Integer(5));
29        System.out.println(":: Object coming back of type : "+n.
              getClass().getName());
30
31        System.out.println(":: Computed sc.compute(5)="+n.toString());
32 //
33 //
34 //          Fractal.getLifeCycleController(simpleComponent).stopFc()
       ;
35 //       System.out.println(":: Component stopped ...");
36
37        System.exit(0);
38
39      } catch (Exception e) {
40        e.printStackTrace();
41        System.exit(0);
42      }
43
44    }
45
46 }
```

The code looks like the same of Listing 6.4. In particular, there are two things that must be observed:

- the composite is created using the same `Factory`, simply passing the `Composite.fractal` descriptor as a parameter
- the sub-component are automatically created by the `Factory` upon discovering they are needed parsing the `Composite.fractal` descriptor.

These two things allow the programmer to perceive the composite assembly as a normal component. In case you were "selling" the composite, you should just provide the descriptor and the user could even not know that there is a composite inside.

Overall, the single component example and the composite component example just discussed should give a precise idea of component definition and usage

within ProActive. Further information related to the descriptors can be found at [1], while further info on components and component sample code can be found at [2].

6.1.3 Sample usage: composite component with deployment

The example in the previous sections shows how a composite can be run using a proper factory and an XML component assembly descriptor. Both the components of the assembly are run in the current JVM, however, i.e. in the JVM where the launcher Java code (Main.java) was run.

Here we'll show how components can be run in different configurations (different JVM on the same host, different JVMs on different hosts) using the *deployment* features provided by GCM/ProActive.

First of all, let us introduce the deployment descriptor. The deployment descriptor host the information relative to the *virtual nodes* used to run the components. A virtual node is basically a name definition paired with a *mapping* to a JVM running on a given host. The JVM can be forked as a consequence of the component instantiation or it can be `acquired` among the already running JVMs on that host.

First of all, we'll show how a component can be mapped on a virtual node (we will use here the `SimpleComponent` and `SecondComponent` of the previous Section).

In order to map a component onto a virtual node, we need to specify a `virtual-node` tag in the component descriptor `.fractal` file. We can therefore modify the `SimpleComponent.fractal` file as follows:

Listing 6.11: SimpleComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
    EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
     proactive.dtd">
4
5 <definition name="vnes1.SimpleComponent">
6
7   <interface
8     name="simple-comp-interface"
9     signature="vnes1.SimpleComponentInterface"
10    role="server" />
11
12  <content
13    class="vnes1.SimpleComponent" />
14
15  <controller
16    desc="primitive" />
17
18  <virtual-node name="simple-component-node" cardinality="single" />
19 </definition>
```

Line 20 is the "new entry" in the file. It states that the component should be eventually instantiated on the `simple-component-node`. In a similar way, we can modify the `SecondComponent.fractal` descriptor to map that component onto a `second-component-node` virtual node.

The we should provide a *deployment descriptor* file. The descriptor file is and XML file that contains:

- tags to initialize property variables
- tags to define virtual nodes
- tags to map virtual nodes to jvms
- tags to instruct the ProActive/GCM environment on the procedures to set up the "infrastructure" of JVMs needed to implement the virtual nodes.

In our case, for the moment, we decide to omit the most interesting features of the infrastructure part, to concentrate on the component to virtual node mapping.

Therefore, we can prepare a deployment descriptor such as the following:

Listing 6.12: vnes1descr.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="urn:proactive:deployment:3.3
5   http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
        deployment.xsd">
6
7    <variables>
8      <descriptorVariable name="PROACTIVE_HOME" value="/Users/
           marcodanelutto/Documents/Didattica/CCCP/Strumenti/ProActive/
           ProActive-4.0.2" />
9      <descriptorVariable name="JAVA_HOME" value="/usr/bin" />
10
11   </variables>
12
13   <componentDefinition>
14     <virtualNodesDefinition>
15       <virtualNode name="second-component-node"/>
16       <virtualNode name="simple-component-node"/>
17     </virtualNodesDefinition>
18   </componentDefinition>
19
20   <deployment>
21
22     <mapping>
23       <map virtualNode="second-component-node">
24          <jvmSet><currentJVM/></jvmSet>
25       </map>
26       <map virtualNode="simple-component-node">
27          <jvmSet><currentJVM/></jvmSet>
28       </map>
29     </mapping>
```

```
30
31   </deployment>
32
33   <infrastructure>
34   </infrastructure>
35
36 </ProActiveDescriptor>
```

The `variables` section of the document, in lines 7 to 9, is just there to show how property variables can be set up.

The `componentDefinition` section defines the virtual nodes we are going to use. Here we defined two virtual nodes with the intent to run the `SimpleComponent` on the `simple-component-node` virtual node and the `SecondComponent` on the `second-component-node` one. These virtual node names are the ones we have to specify in the component `.fractal` descriptor files.

In the `deployment` section, the mapping among the virtual nodes and the JVMs used to run them are defined. Here we use the `currentJVM` tag, for the sake of simplicity. Later we'll show how to instantiate new JVMs.

In the `infrastructure` part the way used to setup the JVMs used in the `deployment` part is described. In this case, we leave it empty, but the the `infrastructure` tags are needed anyway. If we omit this section, we'll get run time exceptions when the descriptor is used.

In order to run a GCM component application using the deployment descriptor, we need to perform an additional step, with respect to what we have already seen in the previous example, when component were run within the current JVM without any kind of deployment: we need to read the deployment descriptor and place it in the *context* passed to the Factory to instantiate the components.

Therefore the launcher of our simple component assembly using deployment descriptor, can be written as follows:

Listing 6.13: Test.java

```
1  package vnes1;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  import org.objectweb.fractal.adl.Factory;
7  import org.objectweb.fractal.api.Component;
8  import org.objectweb.fractal.api.control.LifeCycleController;
9  import org.objectweb.fractal.api.factory.GenericFactory;
10 import org.objectweb.fractal.api.type.ComponentType;
11 import org.objectweb.fractal.api.type.InterfaceType;
12 import org.objectweb.fractal.api.type.TypeFactory;
13 import org.objectweb.fractal.util.Fractal;
14 import org.objectweb.proactive.api.PADeployment;
15 import org.objectweb.proactive.core.component.Constants;
16 import org.objectweb.proactive.core.component.ContentDescription;
17 import org.objectweb.proactive.core.component.ControllerDescription;
```

```
18 import org.objectweb.proactive.core.component.factory.
       ProActiveGenericFactory;
19 import org.objectweb.proactive.core.descriptor.data.
       ProActiveDescriptor;
20 import org.objectweb.proactive.core.descriptor.data.VirtualNode;
21 import org.objectweb.proactive.core.node.Node;
22
23 import vnes1.SecondComponentInterface;
24
25 public class Test {
26
27   /**
28    * @param args
29    */
30   public static void main(String[] args) {
31     try {
32       // get the Factory
33       Factory f = (Factory) org.objectweb.proactive.core.component.
             adl.FactoryFactory.getFactory();
34       // create the context
35       Map<String, Object> context = new HashMap<String, Object>();
36       // find the ProActiveDescriptor (this is the one hosting the
             virtual node definition and mapping
37       ProActiveDescriptor deploymentDescriptor =
38         PADeployment.getProactiveDescriptor("file:///Users/
               marcodanelutto/Documents/Ricerca/Muskel/ProActiveCCP/src
               /vnes1/vnes1descr.xml");
39       // add descriptor to the context
40       context.put("deployment-descriptor", deploymentDescriptor);
41       // activate mappings
42       deploymentDescriptor.activateMappings();
43
44 //      Component first = (Component) f.newComponent("vnes1.
       SimpleComponent", context);
45       // now create components
46       Component composite = (Component) f.newComponent("vnes1.
             Composite", context);
47           System.out.println(":: Composite Component created ...");
48
49           Fractal.getLifeCycleController(composite).startFc();
50       System.out.println(":: Component started ...");
51
52
53       SecondComponentInterface sc = ((SecondComponentInterface)
             composite.getFcInterface("outer-comp-interface"));
54       Object n = sc.compute(new Integer(5));
55       System.out.println(":: Object coming back of type : "+n.
             getClass().getName());
56
57       System.out.println(":: Computed sc.compute(5)="+n.toString());
58 //
59 //
60 //          Fractal.getLifeCycleController(simpleComponent).stopFc()
       ;
61 //      System.out.println(":: Component stopped ...");
62
63       System.out.println("Press <enter> to continue");
```

```
64       System.in.read(); // wait for user ok
65       deploymentDescriptor.killall(false); // then kill JVMs used to
              deploy components
66       // and eventually exit
67       System.exit(0);
68
69     } catch (Exception e) {
70       e.printStackTrace();
71   }
72
73   }
74 }
```

Lines 37 to 42 represent the additional part with respect to the launcher in Listing 6.10. Lines 37–38 read the deployment descriptor (here an absolute filename url is used to denote the descriptor). Line 40 adds the descriptor to the context[1]. Eventually, line 41 activates the virtual nodes and the mappings defined in the deployment descriptor. Actually, there is a second addition, with respect to code in Listing 6.10: this line 65, than is used to shutdown all the JVMs created to host components, according to what stated in the deployment descriptor.

Now let's try to make a further step: we consider the possibility to instantiate new JVMs to run the components in our assembly. We therefore modify the deployment descriptor as follows:

Listing 6.14: vnes1descr-multipleJVM.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="urn:proactive:deployment:3.3
5  http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
      deployment.xsd">
6
7   <variables>
8     <descriptorVariable name="PROACTIVE_HOME" value="/Users/
         marcodanelutto/Documents/Didattica/CCCP/Strumenti/ProActive/
         ProActive-4.0.2" />
9     <descriptorVariable name="JAVA_HOME" value="/usr/bin" />
10
11   </variables>
12
13   <componentDefinition>
14     <virtualNodesDefinition>
15       <virtualNode name="second-component-node"/>
16       <virtualNode name="simple-component-node"/>
17     </virtualNodesDefinition>
18   </componentDefinition>
19
20   <deployment>
21
```

[1]be careful, the name *has* to be “deployment-descriptor” or you'll get an amount of crazy exceptions at run time ...

```
22     <mapping>
23       <map virtualNode="second-component-node">
24   <jvmSet> <vmName value="jvm-1"/> </jvmSet>
25       </map>
26       <map virtualNode="simple-component-node">
27   <jvmSet> <vmName value="jvm-2"/> </jvmSet>
28       </map>
29     </mapping>
30
31     <jvms>
32       <jvm name="jvm-1"><creation><processReference refid="
           jvmProcess" /></creation></jvm>
33       <jvm name="jvm-2"><creation><processReference refid="
           jvmProcess" /></creation></jvm>
34     </jvms>
35
36   </deployment>
37
38   <infrastructure>
39     <processes>
40       <processDefinition id="jvmProcess">
41         <jvmProcess class="org.objectweb.proactive.core.process.
             JVMNodeProcess"> </jvmProcess>
42       </processDefinition>
43     </processes>
44   </infrastructure>
45
46 </ProActiveDescriptor>
```

Now each one of the two component virtual nodes is mapped onto a different JVM, still on the same machine. The JVM is started on the purpose (`creation` tag at lines 32–33; it could have been an `acquisition` tag instead, to acquire existing JVMs). The process used to start the JVMs is the `JVMNodeProcess` from the ProActive/GCM environment, as defined in the `infrastructure` section of the document.

If we change `vnes1descr.xml` to `vnes1descr-multipleJVM.xml` at line 38 in Listing 6.13, we will get the same (usual and correct) output as before. However, if we run a `ps x` on a separate terminal *before* giving the carriage return leading to the termination of the program, we'll get something as:

Listing 6.15: output2jvm.txt

```
1 dhcp-131-114-3-91:vnes1 marcodanelutto$ ps x
2 ...
3 32335   ??  S      2:13.32 /Applications/TeX/TeXShop.app/Contents/
    MacOS/TeXShop -psn_0_6993579
4 32350   ??  SNs    0:02.61 /System/Library/Frameworks/CoreServices.
    framework/Frameworks/Metadata.framework/Versions/A/Support/m
5 32431   ??  Ss     0:00.16 /sw/bin/pdflatex lab.tex
6 32520   ??  S      0:04.57 /System/Library/Frameworks/JavaVM.
    framework/Versions/1.6/Home/bin/java -Dfractal.provider=org.
    object
7 32521   ??  S      0:02.83 /System/Library/Frameworks/JavaVM.
    framework/Versions/1.6.0/Home/bin/java -cp /Users/marcodanelutto
```

```
      /D
 8 32522   ??  S      0:02.96 /System/Library/Frameworks/JavaVM.
      framework/Versions/1.6.0/Home/bin/java -cp /Users/marcodanelutto
      /D
 9 21492 s000  Ss     0:00.03 login -pf marcodanelutto
10 21493 s000  S+     0:00.07 -bash
11 21549 s001  Ss     0:00.01 login -pf marcodanelutto
12 21550 s001  S      0:01.02 -bash
13 32523 s001  R+     0:00.00 ps x
14 22215 s002  Ss     0:00.03 login -pf marcodanelutto
15 22216 s002  S      0:00.20 -bash
16 32478 s002  S+     0:00.01 more distrib-deployment.xml
17 32272 s003  Ss     0:00.10 login -pf marcodanelutto
18 32273 s003  S+     0:00.05 -bash
19 dhcp-131-114-3-91:vnes1 marcodanelutto$
```

clearly indicating (lines 6–8) that three JVMs are currently in execution: one running the `Test` code and the other two running the two components. In case the `vnes1descr.xml` was used instead, we get something as:

Listing 6.16: output1jvm.txt

```
 1 dhcp-131-114-3-91:vnes1 marcodanelutto$ ps x
 2 ...
 3 32335   ??  S      2:05.28 /Applications/TeX/TeXShop.app/Contents/
      MacOS/TeXShop -psn_0_6993579
 4 32350   ??  SNs    0:02.12 /System/Library/Frameworks/CoreServices.
      framework/Frameworks/Metadata.framework/Versions/A/Support/m
 5 32431   ??  Ss     0:00.16 /sw/bin/pdflatex lab.tex
 6 32445   ??  S      0:04.50 /System/Library/Frameworks/JavaVM.
      framework/Versions/1.6/Home/bin/java -Dfractal.provider=org.
      object
 7 21492 s000  Ss     0:00.03 login -pf marcodanelutto
 8 21493 s000  S+     0:00.07 -bash
 9 21549 s001  Ss     0:00.01 login -pf marcodanelutto
10 21550 s001  S      0:00.97 -bash
11 32451 s001  R+     0:00.00 ps x
12 22215 s002  Ss     0:00.03 login -pf marcodanelutto
13 22216 s002  S      0:00.17 -bash
14 32264 s002  S+     0:00.00 more userfarm.fractal
15 32272 s003  Ss     0:00.10 login -pf marcodanelutto
16 32273 s003  S+     0:00.05 -bash
17 dhcp-131-114-3-91:vnes1 marcodanelutto$
```

where there is just one JVM running for both the `Test` code and the two components in the assembly.

6.1.4 Sample usage: single component with deploymnet on a remote node

Consider the `SimpleComponent` discussed before and suppose we want to use it after being deployed to a remote node.

We need to prepare a deployment descriptor, in addition to the component descriptor. The deployment descriptor should set up the infrastructure needed

to run the component on the remote node. In particular, we have to define the *virtual-node* to be used in the component descriptor and then define all the information necessary to run the run time and the component itself on the remote resource.

The deployment descriptor can be defined as follows:

Listing 6.17: distributed.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="urn:proactive:deployment:3.3
5  http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
      deployment.xsd">
6
7   <variables>
8     <descriptorVariable name="PROACTIVE_HOME" value="/home/marcod/
         ProActive-4.0.2" />
9     <descriptorVariable name="JAVA_HOME" value="/home/marcod/jdk/bin
         " />
10
11   </variables>
12
13   <componentDefinition>
14     <virtualNodesDefinition>
15       <virtualNode name="simple-component-node"/>
16     </virtualNodesDefinition>
17   </componentDefinition>
18
19   <deployment>
20
21     <mapping>
22       <map virtualNode="simple-component-node">
23   <jvmSet> <vmName value="jvm-2"/> </jvmSet>
24       </map>
25     </mapping>
26
27     <jvms>
28       <jvm name="jvm-2"> <creation> <processReference refid="process
           -distributed-2" /> </creation> </jvm>
29     </jvms>
30
31   </deployment>
32
33   <infrastructure>
34     <processes>
35
36       <processDefinition id="process-distributed-2">
37         <sshProcess
38           class="org.objectweb.proactive.core.process.ssh.SSHProcess
               "
39           username="marcod" hostname="u12">
40           <processReference refid="process-remote"/>
41         </sshProcess>
42       </processDefinition>
43
44       <processDefinition id="process-remote">
```

```
45        <jvmProcess class="org.objectweb.proactive.core.process.
              JVMNodeProcess">
46          <classpath>
47            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                ProActive.jar"/>
48            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.
                jar"/>
49            <absolutePath value="${PROACTIVE_HOME}/dist/lib/asm
                -2.2.1.jar"/>
50            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                bouncycastle.jar"/>
51            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                dtdparser.jar"/>
52            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal-
                adl.jar"/>
53            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                javassist.jar"/>
54            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ganymed-
                ssh2-build210.jar"/>
55            <absolutePath value="${PROACTIVE_HOME}/dist/lib/junit
                -4.4.jar"/>
56            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.
                jar"/>
57            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                ow_deployment_scheduling.jar"/>
58            <absolutePath value="${PROACTIVE_HOME}/dist/lib/
                xercesImpl.jar"/>
59
60            <absolutePath value="/home/marcod/cccp/"/>
61          </classpath>
62
63          <javaPath>
64            <absolutePath value="${JAVA_HOME}/java" />
65          </javaPath>
66          <policyFile>
67            <absolutePath value="${PROACTIVE_HOME}/examples/
                proactive.java.policy" />
68          </policyFile>
69          <log4jpropertiesFile>
70                <absolutePath value="${PROACTIVE_HOME}/examples/
                    proactive-log4j" />
71          </log4jpropertiesFile>
72          <jvmParameters>
73            <parameter value="-Dproactive.communication.protocol=
                rmissh"/>
74          </jvmParameters>
75        </jvmProcess>
76      </processDefinition>
77
78    </processes>
79  </infrastructure>
80
81 </ProActiveDescriptor>
```

The lines 7–9 define the variables used then to defined classpaths and absolute paths for the jvm.

The lines 13 to 17 define the virtual nodes used. Here we have a single node, but more than one node can be defined. The name of the virtual node (`simple-component-node`) is the one that has to be used in the component descriptor to associate the virtual node to the component.

The `deployment` section defines the information needed to set up the ProActive/GCM run time on the remote machines. Here we used the `mapping` tag to associate a jvm to the virtual node (`map` tags can be repeated any number of time to map different virtual nodes on different jvms). The `jvms` tag is used to associate logical jvms to the names defined in the `mapping` tag. We could also indicate here all the parameters of the jvm, but using a reference to a `process` tag in the `infrastructure` simplifies the descriptor, as we can use the same reference for different logical jvms. Eventually, the `infrastructure` defines all the parameters needed to run the run time on the remote nodes hosting the virtual nodes:

- again, lines 36–42 defined a process specifying part of the parameters (e.g. the host name) and a reference, in such a way most of the common parameters are included in the refereed process and the variable ones are given there

- lines 44–78 define all the parameters needed to run the run time, including the class name of the run time Main (line 45), the jars to be included in the class path (lines 47–58), the class path to be used to retrieve user classes (line 60), the java executable absolute path (linee 63–65), the policy file location (lines 66–68), the log4j configuration file location (lines 70–72), parameters to be passed to the remote jvm (in this case the protocol to be used to connect remote resource (lines 72–74)).

Once the deployment descriptor has been prepared, the only thing to do is to modify the `SimpleComponent.fractal` descriptor, to state that the component has to be deployed on the virtual node defined in the deployment descriptor. This can be achieved adding a virtual node line at the end of the descriptor:

Listing 6.18: SimpleComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="vnes0.SimpleComponent">
6
7   <interface
8     name="simple-comp-interface"
9     signature="vnes0.SimpleComponentInterface"
10     role="server" />
11
12   <content
13     class="vnes0.SimpleComponent" />
```

```
14
15   <controller
16     desc="primitive" />
17
18   <virtual-node name="simple-component-node" cardinality="single" />
19 </definition>
```

In this case the `virtual-node` tag is used to establish the association between the component described in the descriptor and the virtual node appearing in the deployment file.

Apart from the descriptors, the other thing that changes with respect to the code presented before and used to run the `SimpleComponent` onto the current machine is the `main` launching the component. The first things to do now is to read the deployment descriptor. Then the descriptor is included in the `context` passed to the factory reading the component descriptor and creating the component.

The following code shows all the steps needed to run the component remotely:

Listing 6.19: Main.java

```
1 package vnes0;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 import org.objectweb.fractal.adl.Factory;
7 import org.objectweb.fractal.api.Component;
8 import org.objectweb.fractal.util.Fractal;
9 import org.objectweb.proactive.api.PADeployment;
10 import org.objectweb.proactive.core.descriptor.data.
       ProActiveDescriptor;
11
12 public class Main {
13
14   public static void main(String [] args) {
15
16     try {
17       // get factory to instantiate the component from ADL
18       Factory f = org.objectweb.proactive.core.component.adl.
           FactoryFactory.getFactory();
19       // the has table is used to set up the context
20           Map<String, Object> context = new HashMap<String, Object
               >();
21
22       ProActiveDescriptor deploymentDescriptor =
23         PADeployment.getProactiveDescriptor("vnes0/distributed.xml")
             ;
24       // add descriptor to the context
25       context.put("deployment-descriptor", deploymentDescriptor);
26       // activate mappings
27       deploymentDescriptor.activateMappings();
28       System.err.println("mapping activated\nnow starting component
           deploymnet");
29
```

```
30
31       Component simpleComponent = (Component) f.newComponent("vnes0.
             SimpleComponent", context);
32           System.out.println(":: Component created ...");
33
34           Fractal.getLifeCycleController(simpleComponent).startFc();
35       System.out.println(":: Component started ...");
36
37
38       SimpleComponentInterface sc = ((SimpleComponentInterface)
             simpleComponent.getFcInterface("simple-comp-interface"));
39       Object n = sc.doWork(new Integer(5));
40       System.out.println(":: Object coming back of type : "+n.
             getClass().getName());
41
42       System.out.println(":: Computed sc.doWork(5)="+n.toString());
43
44
45           Fractal.getLifeCycleController(simpleComponent).stopFc();
46       System.out.println(":: Component stopped ...");
47
48       System.exit(0);
49
50     } catch (Exception e) {
51       e.printStackTrace();
52       System.exit(0);
53     }
54
55   }
56
57 }
```

The factory is created as in the former examples (line 18), as well as the `Map context` (line 20). The deployment descriptor is read at lines 22–23. The deployment should be present in one of the directories in the class path or specified through a relative path from within one of such directories. Once read, the deployment descriptor has to be added to the context (line 25) and the mappings hosted in the descriptor must be activated (line 27).

At this point, the code is the same code used when running the component locally.

The difference can be seen in the program output:

Listing 6.20: output

```
1 [marcod@u5 ~/cccp]$ java -Dfractal.provider=org.objectweb.proactive.
      core.component.Fractive -Djava.security.policy=./all.permissions
       vnes0.Main
2  --> This ClassFileServer is listening on port 2026
3 Created a new registry on port 1099
4 ************* Reading deployment descriptor: file:vnes0/distributed.
      xml *******************
5 created VirtualNode name=simple-component-node
6 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
7 mapping activated
```

```
8 now starting component deploymnet
9 **** Starting jvm on 10.0.2.12
10  --> This ClassFileServer is listening on port 2030
11 Detected an existing RMI Registry on port 1099
12 **** Mapping VirtualNode simple-component-node with Node: rmissh
       ://10.0.2.12:1099/simple-component-node122318164 done
13 Generating class : pa.stub.vnes0._StubSimpleComponent
14 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
       _StubJMXNotificationListener
15 :: Component created ...
16 :: Component started ...
17 Generating class : pa.stub.java.lang._StubObject
18 --- doWork computing 5 got task 5:java.lang.Integer
19 :: Object coming back of type : pa.stub.java.lang._StubObject
20 :: Computed sc.doWork(5)=6
21 :: Component stopped ...
22 [marcod@u5 ~/cccp]$
```

In this case, we used a cluster with nodes named u*i*. The program is running on node u5 and the deployment descriptor states the component has to be run on u12. The address of node ui is 10.0.2.i. In fact we see that a JVM is started on node 10.0.2.12 and the component is run within that particular JVM. If we modify the code of the `SimpleComponent` in such a way it prints the name of the host where it is running:

Listing 6.21: SimpleComponent.java

```
1 package vnes0;
2
3 public class SimpleComponent implements SimpleComponentInterface {
4
5   @Override
6   public Object doWork(Object task) {
7               try {
8                       String localMachine1 = java.net.InetAddress.
                            getLocalHost().getCanonicalHostName();
9                       System.err.println(":: Hostname of local
                            machine: " + localMachine1);
10              } catch (Exception e) {
11                      e.printStackTrace();
12              }
13
14     Integer i = (Integer) task;
15     int iv = i.intValue();
16     System.err.println("--- doWork computing "+iv+" got task "+task
           +":"+task.getClass().getName());
17     Integer r = new Integer(++iv);
18     return r;
19   }
20
21 }
```

we will get the following output, where the `SimpleComponent` prints the name of the u12 machine:

Listing 6.22: output1

```
1 [marcod@u5 ~/cccp]$ java -Dfractal.provider=org.objectweb.proactive.
      core.component.Fractive -Djava.security.policy=./all.permissions
       vnes0.Main
2  --> This ClassFileServer is listening on port 2026
3 Created a new registry on port 1099
4 ************* Reading deployment descriptor: file:vnes0/distributed.
      xml ********************
5 created VirtualNode name=simple-component-node
6 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
7 mapping activated
8 now starting component deploymnet
9 **** Starting jvm on 10.0.2.12
10  --> This ClassFileServer is listening on port 2035
11 Detected an existing RMI Registry on port 1099
12 **** Mapping VirtualNode simple-component-node with Node: rmissh
      ://10.0.2.12:1099/simple-component-node1686586965 done
13 Generating class : pa.stub.vnes0._StubSimpleComponent
14 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
15 :: Component created ...
16 :: Component started ...
17 Generating class : pa.stub.java.lang._StubObject
18 :: Object coming back of type : pa.stub.java.lang._StubObject
19 :: Hostname of local machine: u12
20 :: doWork computing 5 got task 5:java.lang.Integer
21 :: Computed sc.doWork(5)=6
22 :: Component stopped ...
23 [marcod@u5 ~/cccp]$
```

It is worth pointing out that the program has been started with the usual command:

```
java -Dfractal.provider=org.objectweb.proactive.core.component.Fractive
          -Djava.security.policy=./all.permissions vnes0.Main
```

specifying both the Fractal provider class and the java security permission file.

6.1.5 Sample usage: composite component with deployment on remote nodes

As the final example, we'll show how the deployment of the composite component assembly can be modified in such a way the part of the components are run on remote nodes.

We'll use the `rmissh` protocol implemented by ProActive/GCM to get rid of the firewall problems, assuming that:

1. the user has access to a remote host with ProActive installed (i.e. he has a valid account on the remote machine) and

2. the account is set up in such a way no passwd is required to issue an ssh command.[2]

In order to be able to run components on a remote machine, we need to change the deployment descriptor. In particular, we need to change the lines relative to the mapping of the components onto the JVMs. Instead of using a <currentJVM/> placement we should use a creation tag and we should associate to the newly created JVM machine a process tag specifying how to the create the remote JVM instance. We should also specify how the remote processing element can be reached, as an example by providing the user name or the paths to ProActive and to the other classes needed.

In order to get a version of the component program discussed in the previous section that runs on a distributed environment, we can use the following deployment descriptor:

Listing 6.23: pianosa.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="urn:proactive:deployment:3.3
5  http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
      deployment.xsd">
6
7   <variables>
8     <descriptorVariable name="REMOTE_PROACTIVE_HOME" value="/home/
        marcod/ProActive-4.0.2" />
9     <descriptorVariable name="REMOTE_JAVA_HOME" value="/home/marcod/
        jdk/bin" />
10
11  </variables>
12
13  <componentDefinition>
14    <virtualNodesDefinition>
15      <virtualNode name="second-component-node"/>
16      <virtualNode name="simple-component-node"/>
17    </virtualNodesDefinition>
18  </componentDefinition>
19
20  <deployment>
21
22    <mapping>
23      <map virtualNode="second-component-node">
24  <jvmSet> <vmName value="jvm-1"/> </jvmSet>
25      </map>
26      <map virtualNode="simple-component-node">
27  <jvmSet> <vmName value="jvm-2"/> </jvmSet>
28      </map>
29    </mapping>
30
```

[2]This can be accomplished by inserting in the $HOME/.ssh/authorized_keys file of the target machine the public key used to negotiate the private key setup in ssh, i.e. the contents of the $HOME/id_rsa.pub (in case we used RSA) file on the machine were the ssh command is launched.

```
31    <jvms>
32      <jvm name="jvm-1"> <creation> <processReference refid="process
            -distributed-1" /> </creation> </jvm>
33      <jvm name="jvm-2"> <creation> <processReference refid="process
            -distributed-2" /> </creation> </jvm>
34    </jvms>
35
36  </deployment>
37
38  <infrastructure>
39    <processes>
40
41      <processDefinition id="process-distributed-1">
42        <sshProcess class="org.objectweb.proactive.core.process.ssh.
              SSHProcess"
43          username="marcod" hostname="u12">
44          <processReference refid="process-remote"/>
45        </sshProcess>
46      </processDefinition>
47
48      <processDefinition id="process-distributed-2">
49        <sshProcess
50          class="org.objectweb.proactive.core.process.ssh.SSHProcess
                "
51          username="marcod"
52          hostname="u13">
53          <processReference refid="process-remote"/>
54        </sshProcess>
55      </processDefinition>
56
57      <processDefinition id="process-remote">
58        <jvmProcess class="org.objectweb.proactive.core.process.
              JVMNodeProcess">
59          <classpath>
60            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  ProActive.jar"/>
61            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  fractal.jar"/>
62            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  asm-2.2.1.jar"/>
63            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  bouncycastle.jar"/>
64            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  dtdparser.jar"/>
65            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  fractal-adl.jar"/>
66            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  javassist.jar"/>
67            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  ganymed-ssh2-build210.jar"/>
68            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  junit-4.4.jar"/>
69            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  log4j.jar"/>
70            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
                  ow_deployment_scheduling.jar"/>
71            <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
```

```
                 xercesImpl.jar"/>
72
73             <absolutePath value="/home/marcod/cccp"/>
74           </classpath>
75
76           <javaPath>
77             <absolutePath value="${REMOTE_JAVA_HOME}/java" />
78           </javaPath>
79           <policyFile>
80             <absolutePath value="${REMOTE_PROACTIVE_HOME}/examples/
                 proactive.java.policy" />
81           </policyFile>
82           <log4jpropertiesFile>
83                 <absolutePath value="${REMOTE_PROACTIVE_HOME}/
                     examples/proactive-log4j" />
84           </log4jpropertiesFile>
85           <jvmParameters>
86             <parameter value="-Dproactive.communication.protocol=
                 rmissh"/>
87           </jvmParameters>
88         </jvmProcess>
89       </processDefinition>
90
91     </processes>
92   </infrastructure>
93
94 </ProActiveDescriptor>
```

At lines 7–11 we define some variables that will be used in the deployment. The important part is the definition of the virtual nodes, obviously. Two virtual node names are defined at lines 13–18. Then in the deployment section (lines 22–29) we map the two virtual nodes on two virtual machines: jvm-1 and jvm-2 . These JVM are used to run a process-distributed-1 and a process-distributed-2 process, as specified at lines 32 and 33. In turn, this process is defined (lines 48–52) as a process run through ssh tunnelling on host marcod.homenet.telecomitalia.it. The process-remote run within process-distributed-i are defined with lines 30–79. Here we defined:

- the class used to run the run time on the node (this is the standard SSHProcess provided by the ProActive run time (line 58))
- the classpath, including both the ProActive run time and the component files (lines 59–74)
- the path to Java interpreter, policy and logging property files (lines 76–84)
- the parameters passed to the remote JVM (lines 85–88). These are important as with the parameters we also define the communication protocol to be used, in this case the RMI tunnelling through ssh.

It is particularly important to point out that the variables defined in the initial part of the deployment file can be used to customize the remote execution of components, that can be run on machines with different locations

of the java executables, of the ProActive directory and of the Component files. By consulting the online documentation of ProActive/GCM deploymnet files, you can easily discover that the needed files can be staged to the remote machines using scp simply by specifying proper `fileTransferDeploy` and `fileTransferRetrieve` tags.

When running out two component assembly using this deployment descriptor, we'll get the following output:

Listing 6.24: output

```
1 [marcod@u5 ~/cccp]$ java -Dfractal.provider=org.objectweb.proactive.
      core.component.Fractive -Djava.security.policy=./all.permissions
       vnes1.Test
2 :: GOT FACTORY
3 :: CREATED CONTEXT MAP
4  --> This ClassFileServer is listening on port 2026
5 Created a new registry on port 1099
6 ************** Reading deployment descriptor: file:///home/marcod/
      cccp/vnes1/pianosa.xml ********************
7 created VirtualNode name=second-component-node
8 created VirtualNode name=simple-component-node
9 :: GOT PROACTIVE DESCRIPTOR
10 :: PUT DESCRIPTOR IN CONTEXT
11 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
12 :: ACTIVATED MAPPINGS
13 mapping activated
14 now starting component deploymnet
15 **** Starting jvm on 10.0.2.13
16 **** Starting jvm on 10.0.2.12
17  --> This ClassFileServer is listening on port 2026
18 Created a new registry on port 1099
19  --> This ClassFileServer is listening on port 2026
20 Created a new registry on port 1099
21 **** Mapping VirtualNode simple-component-node with Node: rmissh
      ://10.0.2.13:1099/simple-component-node617019055 done
22 **** Mapping VirtualNode second-component-node with Node: rmissh
      ://10.0.2.12:1099/second-component-node860041102 done
23 Generating class : pa.stub.vnes1._StubSecondComponent
24 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
25 Generating class : pa.stub.vnes1._StubSimpleComponent
26 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
27 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
28 :: Composite Component created ...
29 :: Component started ...
30 :: SecondComponent :: Hostname of local machine: u12
31 :: SimpleComponent :: Hostname of local machine: u13
32 --- doWork computing 5 got task 5:java.lang.Integer
33 :: hostname: i386
34 --- called SimpleComponent, got 6 type java.lang.Integer
35 :: Object coming back of type : java.lang.Integer
36 :: Computed sc.compute(5)=12
37 Press <enter> to continue
38
```

```
39 node simple-component-node617019055 is being killed, terminating
       body -338392b1-1211190e585--7fee--3e84b5785f29789d--338392b1
       -1211190e585--8000
40 node Node1366617550 is being killed, terminating body -338392b1
       -1211190e585--7fc4--3e84b5785f29789d--338392b1-1211190e585--8000
41 terminating Runtime //10.0.2.13/PA_JVM1837501017
42 Process finished Thread=ERR -> ssh -l marcod u13  -
43 unable to contact remote object at rmissh://10.0.2.13:1099/
       PA_JVM1837501017 when calling killRT
44  Virtual Machine 3e84b5785f29789d:-338392b1:1211190e585:-8000 on
        host 10.0.2.13 terminated.
45 The unsubscribe action has failed : The objectName=org.objectweb.
       proactive.core.runtimes:type=Runtime,runtimeUrl=rmi
       -//10.0.2.5-1099/PA_JVM464100176 has been already unsubscribe
46 Process finished Thread=IN -> ssh -l marcod u13  -
47 node second-component-node860041102 is being killed, terminating
       body 1d9792f6-12111878338--7fee--a9ef8807ca30c7ef-1d9792f6
       -12111878338--8000
48 node Node272236384 is being killed, terminating body 1d9792f6
       -12111878338--7fc4--a9ef8807ca30c7ef-1d9792f6-12111878338--8000
49 terminating Runtime //10.0.2.12/PA_JVM1723678229
50 unable to contact remote object at rmissh://10.0.2.12:1099/
       PA_JVM1723678229 when calling killRT
51  Virtual Machine a9ef8807ca30c7ef:1d9792f6:12111878338:-8000 on host
         10.0.2.12 terminated.
52 The unsubscribe action has failed : The objectName=org.objectweb.
       proactive.core.runtimes:type=Runtime,runtimeUrl=rmi
       -//10.0.2.5-1099/PA_JVM464100176 has been already unsubscribe
53 Process finished Thread=IN -> ssh -l marcod u12  -
54 Process finished Thread=ERR -> ssh -l marcod u12  -
55 [marcod@u5 ~/cccp]$
```

6.1.6 Sample usage: one-to-many communications through collective interfaces

GCM introduced the concept of collective interfaces. A collective interface is a mechanism implementing communications among a single port and a *collection* of ports (or viceversa). In this Section we show how to implement a one-to-many scatter communication pattern. One component has a port delivering collectively data to a collection of ports. The data to be sent is a `List<T>` and the data actually sent to each one of the ports in the collection is will have type `T`. This relationship between data at the two sides of the communication is fixed, in that is the only one supported when implementing a scatter patter (e.g. a pattern were part of the original data is sent to each one of the ports in the target collection). In the next Section, we'll see how to implement a broadcast operation instead. In that case, the type of data transmitted and the type of data received at the single port in the target collection will coincide.

The scenario we discuss in this section is the following (see Fig. 6.2:

- a `MasterComponent` is provided to the final user, exporting a server (i.e. provides) port with type `List<Integer> doWork(List<Integer>)`

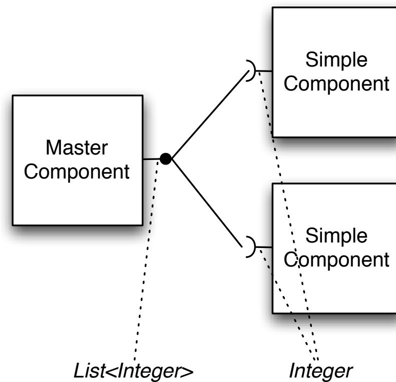

Figure 6.2: Sample configuration with two components connected to a master through a multicast port

The user calling this port with a list of integers will receive back a list of results representing the computation of a function f on each one of the integers in the list

- the function f is implemented in a `SimpleComponent`. Two `SimpleComponent` components will be included in the composite assembly presented to the user. These components will interact with the `MasterComponent` by means of a collective interface.
- the collective interface connecting the `MasterComponent` to the two `SimpleComponent` will be a *multicast* interface (in GCM terminology), that is a port connecting a single client (use) port to a collection of server (provide) ports.

In order to use a collective interface to implement a scatter, we should basically:

- properly describe the port in the component descriptor files (the `.fractal` files)
- properly describe the methods used to implement the component interfaces in the Java code

To use a multicast collective port, it has to be defined as a `client` port with `cardinality="multicast"` in the `fractal` component descriptors. Then, properly typed `server` ports have to be defined on the target collection components. Eventually, multiple bindings have to be established in the composite component descriptor, one per target port in the collection, binding the unique client (`multicast`) port to each one of the target `server` ports.

In our example, we will use the already defined `SimpleComponent` component. The descriptor files of the components used in the composite will therefore be the following.

Listing 6.25: SimpleComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="collective.adl.SimpleComponent">
6
7   <interface
8     name="simple-component-interface"
9     signature="collective.SimpleComponentInterface"
10    role="server" />
11
12  <content
13    class="collective.SimpleComponent" />
14
15  <controller
16    desc="primitive" />
17
18  <!-- <virtual-node name="simple-component-node" cardinality="
        single" /> -->
19 </definition>
```

In the `SimpleComponent` descriptor there is nothing new with respect to the descriptors we already know.

Listing 6.26: MasterComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="collective.adl.MasterComponent">
6
7   <interface
8     name="master-component-interface"
9     signature="collective.MasterComponentInterface"
10    role="server" />
11
12  <interface
13    name="simple-component-interface"
14    signature="collective.SimpleComponentMulticastInterface"
15    role="client"
16  cardinality="multicast"/>
17
18  <content
19    class="collective.MasterComponent" />
20
21  <controller desc="primitive"/>
```

```
22   <!-- <virtual-node name="master-component-node" cardinality="
         single" /> -->
23
24 </definition>
```

In the MasterComponent descriptor, at lines 12–16 we define the collective use port and the cardinality is set to multicast. This is needed to enable the run time to distinguish the port and to handle consequently the data types of the parameters used that otherwise do not match. Also, please pay attention to the signature used to denote the client multicast interface: this is a SimpleComponentMulticastInterface rather than the plain SimpleComponentInterface we already used to denote the server interface of SimpleComponent. In fact, the SimpleComponentInterface exposes a port of type Integer doWork(Integer task) that does not match the type List<Integer> doWork(List<Integer> task) exposed by the MasterComponent client mutlicast port (see below the Java code).

Listing 6.27: CollectiveComposite.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
     EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
      proactive.dtd">
4
5 <definition name="collective.adl.CollectiveComposite">
6
7   <interface
8     name="outer-component-interface"
9     signature="collective.MasterComponentInterface"
10    role="server" />
11
12   <component
13     name="master-component"
14     definition="collective.adl.MasterComponent" />
15
16   <component
17     name="simple-component-1"
18     definition="collective.adl.SimpleComponent" />
19
20   <component
21     name="simple-component-2"
22     definition="collective.adl.SimpleComponent" />
23
24   <binding
25     client="master-component.simple-component-interface"
26     server="simple-component-1.simple-component-interface" />
27   <binding
28     client="master-component.simple-component-interface"
29     server="simple-component-2.simple-component-interface" />
30
31   <binding
32     client="this.outer-component-interface"
33     server="master-component.master-component-interface" />
34
```

```
35   <controller desc="composite"/>
36
37 </definition>
```

In the `CollectiveComposite` descriptor, at lines 24–29, we set up one binding the target collection component between the `MasterComponent` client port and the target component server port.

As usual, the two descriptors for the `MasterComponent` and `SimpleComponent` have a `controller desc="primitive"` while the composite component has a `controller desc="composite"`.

Let us look at the Java code now. The code for the `SimpleComponent` and `SimpleComponentInterface` are the usual ones:

Listing 6.28: SimpleComponent.java

```
1 package collective;
2
3 import java.net.UnknownHostException;
4
5 public class SimpleComponent implements SimpleComponentInterface {
6
7   @Override
8   public Integer doWork(Integer task) {
9     int iv = task.intValue();
10    System.err.println(":: doWork computing "+iv+" at "+this+" got
          task "+task+":"+task.getClass().getName());
11    return new Integer(++iv);
12  }
13
14 }
```

Listing 6.29: SimpleComponentInterface.java

```
1 package collective;
2
3 public interface SimpleComponentInterface {
4
5   public Integer doWork(Integer task);
6 }
```

The code for the `MasterComponent` is a little bit more complicate. It is a component exposing also client ports and therefore we need to implement the BindingController interface. The attribute hosting the reference to the component bind to the client interface has a type which is not the `SimpleComponentInterface`, however. It is a different type:

Listing 6.30: SimpleComponentMulticastinterface.java

```
1 package collective;
2
3 import java.util.List;
4
5 import org.objectweb.proactive.core.component.type.annotations.
      multicast.ClassDispatchMetadata;
```

```
6 import org.objectweb.proactive.core.component.type.annotations.
     multicast.MethodDispatchMetadata;
7 import org.objectweb.proactive.core.component.type.annotations.
     multicast.ParamDispatchMetadata;
8 import org.objectweb.proactive.core.component.type.annotations.
     multicast.ParamDispatchMode;
9
10 @ClassDispatchMetadata(mode = @ParamDispatchMetadata(mode =
     ParamDispatchMode.ROUND_ROBIN))
11 public interface SimpleComponentMulticastInterface {
12
13     @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode =
          ParamDispatchMode.ROUND_ROBIN))
14   public List<Integer> doWork(
15       @ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN)
16       List<Integer> tasks);
17
18 }
```

This exposes an interface which represents the "collective" types (`List<Integer>` in our case) properly annotated to define the way this can be handled to match the actual `SimpleComponent` interface that exposes the non collective types (`Integer` in our case). We used here annotations at the class, method and parameter level. Parameter level annotations overwrite class and method level annotations, while method level annotations overwrite class level ones. Actually, as we use here the very same modes, we could have used only the class level annotation (the first one), with the very same semantics. We indicated all the three just for exposing the correct syntax.

We used here the `ROUND_ROBIN` mode. This mode schedules in a round robin way the elements of the list of integers to the server ports in the target server port collection. Alternatives could have been to use the `ONE_TO_ONE` that in addition pretends to have a number of target server ports equal to the length of the input list[3].

With these assumptions, the code of the `MasterComponent` is as follows:

Listing 6.31: MasterComponent.java

```
1 package collective;
2
3 import java.util.List;
4
5 import org.objectweb.fractal.api.NoSuchInterfaceException;
6 import org.objectweb.fractal.api.control.BindingController;
7 import org.objectweb.fractal.api.control.IllegalBindingException;
8 import org.objectweb.fractal.api.control.IllegalLifeCycleException;
9
10 public class MasterComponent implements BindingController,
     MasterComponentInterface {
11   /**
12    * this is the varaible binded through the binding controller at
          deployment time, when the ADL of the composite component is
```

[3]other modes are possibile, not implementing the scatter policy, such as the `MULTICAST` and `BROADCAST` one

```
        processed
13     */
14    SimpleComponentMulticastInterface slaves = null;
15
16    /**
17     * constructor(void) needed for serialization
18     */
19    public MasterComponent() {}
20
21    /**
22     * interface to be promoted as composite component service
23     * @return
24     */
25    public List<Integer> doWork(List<Integer> tasks) {
26      List<Integer> res = slaves.doWork(tasks);
27      return res;
28    }
29
30    /**
31     * methods require because of the composite (Binding interface)
32     */
33
34    @Override
35    public void bindFc(String clientItfName, Object serverItf)
36        throws NoSuchInterfaceException, IllegalBindingException,
37        IllegalLifeCycleException {
38      if(clientItfName.equals("simple-component-interface")) {
39        slaves = (SimpleComponentMulticastInterface) serverItf;
40      } else {
41        throw new NoSuchInterfaceException(clientItfName);
42      }
43    }
44
45    @Override
46    public String[] listFc() {
47      String[] s =  {"simple-component-interface"};
48      return s;
49    }
50
51    @Override
52    public Object lookupFc(String clientItfName) throws
        NoSuchInterfaceException {
53      if(clientItfName.equals("simple-component-interface"))
54        return slaves;
55      else
56        throw new NoSuchInterfaceException(clientItfName);
57    }
58
59    @Override
60    public void unbindFc(String clientItfName) throws
        NoSuchInterfaceException,
61        IllegalBindingException, IllegalLifeCycleException {
62      if(clientItfName.equals("simple-component-interface"))
63        slaves = null;
64      else
65        throw new NoSuchInterfaceException(clientItfName);
66    }
```

```
67
68 }
```

At line 14, we declare the attribute needed to host the reference to the server ports bind to the component use port. Lines 25–27 implement the actual service exposed by the component on its server port (promoted to composite server port in the composite assembly `fractal` file, listing 6.27). Here we get the input list of integers and we simply invoke the method defined in the `SimpleComponent` as a server port, with the parameters (parameter types) defined in the annotated multicast interface (listing 6.30).

Important: the names of the client and of the server ports used in a collective GCM port *must be the same*. The ProActive GCM runtime will raise exceptions in case the client port is named differently from the server port.

To complete the example, we will use a simple client code:

Listing 6.32: Test.java

```
1 package collective;
2
3 import java.util.ArrayList;
4 import java.util.HashMap;
5 import java.util.Iterator;
6 import java.util.List;
7 import java.util.Map;
8
9 import org.objectweb.fractal.adl.Factory;
10 import org.objectweb.fractal.api.Component;
11 import org.objectweb.fractal.util.Fractal;
12
13 import adles2.SecondComponentInterface;
14
15 public class Test {
16
17   /**
18    * @param args
19    */
20   public static void main(String[] args) {
21     try {
22       // get factory to instantiate the component from ADL
23       Factory f = org.objectweb.proactive.core.component.adl.
              FactoryFactory.getFactory();
24       // the has table is used to set up the context
25       Map<String, Object> context = new HashMap<String, Object>();
26
27       Component composite = (Component) f.newComponent("collective.
              adl.CollectiveComposite", context);
28       System.out.println(":: Composite Component created ...");
29
30       Fractal.getLifeCycleController(composite).startFc();
31       System.out.println(":: Component started ...");
32
33
34       MasterComponentInterface sc = ((MasterComponentInterface)
              composite.getFcInterface("outer-component-interface"));
35
```

```
36       List<Integer> params = new ArrayList<Integer>();
37       for(int i=0; i<8; i++)
38         params.add(new Integer(i));
39 
40       System.err.print(":: list of params is = <");
41       Iterator<Integer> it = params.iterator();
42       while(it.hasNext())
43         System.err.print(it.next()+":");
44       System.err.println(">");
45 
46       List<Integer> res = sc.doWork(params);
47 
48       System.err.println(":: computed results with multicast nodes
             ..."+res);
49 
50       Iterator<Integer> itres = res.iterator();
51       while(itres.hasNext()) {
52         System.err.println(":: got result ->> "+itres.next());
53       }
54       System.exit(0);
55 
56     } catch (Exception e) {
57       e.printStackTrace();
58       System.exit(0);
59     }
60 
61   }
62 
63 }
```

The code obtains a factory (line 23), loads the composite descriptor (line 27), starts the composite assembly invoking the proper service of the lifecycle controller (line 30), gets the reference to the component server interface (line 34) and eventually invokes that service just once (line 46).

The component descriptor files in this case do not include any virtual node, nor we used the deployment facilities of ProActive/GCM, therefore the whole program can be run on a single machine. In this case, the output obtained will be something as:

Listing 6.33: collective.output.txt

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.collective._StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
      _StubProActiveInterfaceImpl
6 Generating class : pa.stub.collective._StubSimpleComponent
7 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
8 :: Composite Component created ...
9 :: Component started ...
10 :: list of params is = <0:1:2:3:4:5:6:Generating class : pa.stub.
      java.util._StubList
11 7:>
```

```
12 :: doWork computing 1 at collective.SimpleComponent@3c8d3e36 got
      task 1:java.lang.Integer
13 :: doWork computing 3 at collective.SimpleComponent@3c8d3e36 got
      task 3:java.lang.Integer
14 :: doWork computing 0 at collective.SimpleComponent@7d829c18 got
      task 0:java.lang.Integer
15 :: doWork computing 2 at collective.SimpleComponent@7d829c18 got
      task 2:java.lang.Integer
16 :: doWork computing 4 at collective.SimpleComponent@7d829c18 got
      task 4:java.lang.Integer
17 :: doWork computing 6 at collective.SimpleComponent@7d829c18 got
      task 6:java.lang.Integer
18 :: doWork computing 5 at collective.SimpleComponent@3c8d3e36 got
      task 5:java.lang.Integer
19 :: doWork computing 7 at collective.SimpleComponent@3c8d3e36 got
      task 7:java.lang.Integer
20 :: computed results with multicast nodes ...org.objectweb.proactive.
      core.group.ProxyForGroup@6b4b5785
21 :: got result ->> 1
22 :: got result ->> 2
23 :: got result ->> 3
24 :: got result ->> 4
25 :: got result ->> 5
26 :: got result ->> 6
27 :: got result ->> 7
28 :: got result ->> 8
```

Lines from 12 to 19 are output by the two `SimpleComponent` instances. The two instances, being run in the same JVM as the `Test` program, simply have a different thread id (the `collective.SimpleComponent@3c8d3e36` and `collective.SimpleComponent@7d829c18` in these lines) showing that actually the two component instances got the single tasks to be computed according to the round robin scheduling policy stated in Listing 6.30.

6.1.7 Sample usage: broadcast with collective interfaces

Let us consider a slight variation of the example discussed in the previous Section. Let us investigate what's needed to implement a similar component assembly where actually the whole input data passed to the server interface exposed to the user is broadcasted to all the server interfaces in the target component collection.

First, we should modify the `SimpleComponent` (and therefore the associated `SimpleComponentInterface`) to reflect the fact the input type is now a `List<Intger>` rather than an `Integer`. We keep the same output type however, and therefore the `SimpleComponent` is modified to compute the length of the list it receives as input:

Listing 6.34: SimpleComponent.java

```
1 package collectiveBroadcast;
2
3 import java.net.UnknownHostException;
4 import java.util.Iterator;
```

```
 5 import java.util.List;
 6
 7 public class SimpleComponent implements SimpleComponentInterface {
 8
 9   @Override
10   public Integer doWork(List<Integer> task) {
11     Iterator<Integer> it = task.iterator();
12     int n = 0;
13     System.err.println(":: worker "+this+" got "+task.getClass().
           getName());
14     while(it.hasNext()) {
15       System.err.println(":: worker "+this+" got : "+it.next());
16       n++;
17     }
18     return n;
19   }
20
21 }
```

Listing 6.35: SimpleComponentInterface.java

```
1 package collectiveBroadcast;
2
3 import java.util.List;
4
5 public interface SimpleComponentInterface {
6
7   public Integer doWork(List<Integer> task);
8 }
```

Then we have to modify the policies used to distribute data to the server port collection, i.e. the `SimpleComponentMulticastInterface.java` file:

Listing 6.36: SimpleComponentMulticastInterface.java

```
 1 package collectiveBroadcast;
 2
 3 import java.util.List;
 4
 5 import org.objectweb.proactive.core.component.type.annotations.
       multicast.ClassDispatchMetadata;
 6 import org.objectweb.proactive.core.component.type.annotations.
       multicast.MethodDispatchMetadata;
 7 import org.objectweb.proactive.core.component.type.annotations.
       multicast.ParamDispatchMetadata;
 8 import org.objectweb.proactive.core.component.type.annotations.
       multicast.ParamDispatchMode;
 9
10 @ClassDispatchMetadata(mode = @ParamDispatchMetadata(mode =
       ParamDispatchMode.BROADCAST))
11 public interface SimpleComponentMulticastInterface {
12
13     // @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode =
           ParamDispatchMode.BROADCAST))
14   public List<Integer> doWork(
15       // @ParamDispatchMetadata(mode = ParamDispatchMode.BROADCAST)
16       List<Integer> tasks);
```

```
17
18 }
```

Here we use the class annotation to say all the input parameters to the server ports of the collection are `BROADCAST`.

These are the only modifications needed to both the Java and the Fractal code in the example. If we run the `Test` program again, we'll get an output such as:

Listing 6.37: collectiveBroadcast.output.txt

```
1 --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.collectiveBroadcast._StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
      _StubProActiveInterfaceImpl
6 Generating class : pa.stub.collectiveBroadcast._StubSimpleComponent
7 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
8 :: Composite Component created ...
9 :: Component started ...
10 :: list of params is = <Generating class : pa.stub.java.util.
      _StubList
11 0:1:2:3:4:5:6:7:>
12 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got java.util
      .ArrayList
13 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 0
14 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 1
15 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 2
16 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 3
17 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 4
18 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 5
19 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 6
20 :: worker collectiveBroadcast.SimpleComponent@561f2f76 got : 7
21 :: worker collectiveBroadcast.SimpleComponent@520f2037 got java.util
      .ArrayList
22 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 0
23 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 1
24 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 2
25 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 3
26 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 4
27 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 5
28 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 6
29 :: worker collectiveBroadcast.SimpleComponent@520f2037 got : 7
30 :: computed results with multicast nodes ...org.objectweb.proactive.
      core.group.ProxyForGroup@4c1
31 :: got result ->> 8
32 :: got result ->> 8
```

Line 13–21 and 22–30 show that both `SimpleComponent` component instances receive the whole list and lines 31–32 show the results list has one item per `SimpelComponent` component instance, as expected.

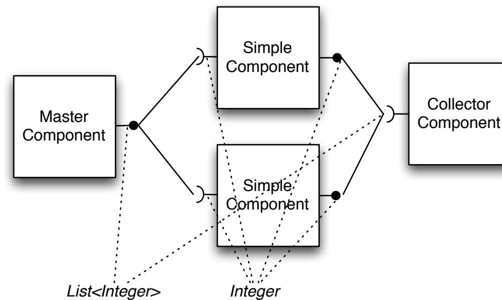

Figure 6.3: Sample configuration with two components connected to a master and a collector through a multicast and a gathercast port

6.1.8 Sample usage: gathercast collective interface

In the previous sections, we used collective interfaces to dispatch tasks to a collection of similar components. The interaction between the `MasterComponent` and the `SimpleComponent` instances is completely based on the RMI/RPC mechanism: the `MasterComponent` wrapping implemented by ProActive/GCM invokes server ports on the `SimpleComponent` instances and gets back, as the result of the invocation, some result data.

Here, we want to investigate how a different interaction can be implemented, with the master sending input data to the simple component instances, and with these component instances delivering result data to a "collector" component as outlined in Fig. 6.3.

In order to define a gathercast[4] we follow a dual process with respect to the one followed to implement a multicast interface.

- we define all the components, taking care of having a client port on the `SimpleComponent` with type `T` to be connected to a server port on the `CollectorComponent` with type `List<Integer>`
- we denote the type of the server port in the `CollectorComponent.fractal` descriptor file as `cardinality="gathercast"` and the client port in the `SimpleComponent.fractal` descriptor file as `cardinality=singleton` (this is important as well, otherwise the whole things does not work).
- we use a java interface `CollectorGathercastInterface` to set up the signature of the `SimpleComponent` client interface in such a way the types match, as we did with the `SimpleComponentMulticastInterface` for the one-2-many multicast interface[5].

[4]the term gathercast is ProActive/GCM jargon: multicast was used to denote the one-2-N interfaces and gathercast (gather as opposite to multi) has been used to denote the N-2-one interfaces

[5]the fact we used the names compound with the `Multicast` or `Gathercast` terms has no

Therefore we'll use the following component code:

Listing 6.38: SimpleComponent.java

```
1 package completeOneWay;
2
3 import java.net.UnknownHostException;
4
5 import org.objectweb.fractal.api.NoSuchInterfaceException;
6 import org.objectweb.fractal.api.control.BindingController;
7 import org.objectweb.fractal.api.control.IllegalBindingException;
8 import org.objectweb.fractal.api.control.IllegalLifeCycleException;
9
10 import collective.SimpleComponentMulticastInterface;
11
12 public class SimpleComponent implements SimpleComponentInterface,
       BindingController {
13
14   CollectorGathercastInterface collector = null;
15
16   @Override
17   public void doWork(Integer task) {
18     int iv = task.intValue();
19     System.err.println(":: doWork computing "+iv+" at "+this+" got
           task "+task+":"+task.getClass().getName());
20     Integer res = new Integer(++iv);
21     try {
22       Thread.sleep(2000);
23     } catch (InterruptedException e) {
24       System.err.println(":: Interrupted ");
25     }
26     collector.deliver(res);
27     System.err.println(":: doWork delivered! (at "+this+")");
28     return;
29   }
30
31   @Override
32   public void bindFc(String clientItfName, Object serverItf)
33   throws NoSuchInterfaceException, IllegalBindingException,
34   IllegalLifeCycleException {
35     if(clientItfName.equals("collector-component-interface")) {
36       collector = (CollectorGathercastInterface) serverItf;
37     } else {
38       throw new NoSuchInterfaceException(clientItfName);
39     }
40   }
41
42
43   @Override
44   public String[] listFc() {
45     String[] s =  {"simple-component-interface"};
46     return s;
47   }
48
49   @Override
```

importance. We could have named the `SimpleComponentMulticastInterface` `Foo` and things should have worked anyway

```
50   public Object lookupFc(String clientItfName) throws
         NoSuchInterfaceException {
51     if(clientItfName.equals("collector-component-interface"))
52       return collector;
53     else
54       throw new NoSuchInterfaceException(clientItfName);
55   }
56
57   @Override
58   public void unbindFc(String clientItfName) throws
         NoSuchInterfaceException,
59   IllegalBindingException, IllegalLifeCycleException {
60     if(clientItfName.equals("colllector-component-interface"))
61       collector = null;
62     else
63       throw new NoSuchInterfaceException(clientItfName);
64   }
65
66 }
```

The `SimpleComponent` implements now the `BindingController` as it also has client ports, in addition to the server ports. The local computation actually ends delivering the result to the collector gathercast port (`deliver` at line 26).

Listing 6.39: MasterComponent.java

```
1 package completeOneWay;
2
3 import java.util.List;
4
5 import org.objectweb.fractal.api.NoSuchInterfaceException;
6 import org.objectweb.fractal.api.control.BindingController;
7 import org.objectweb.fractal.api.control.IllegalBindingException;
8 import org.objectweb.fractal.api.control.IllegalLifeCycleException;
9
10 public class MasterComponent implements BindingController,
       MasterComponentInterface {
11   /**
12    * this is the varaible binded through the binding controller at
          deployment time, when the ADL of the composite component is
          processed
13    */
14   SimpleComponentMulticastInterface slaves = null;
15
16   /**
17    * constructor(void) needed for serialization
18    */
19   public MasterComponent() {}
20
21   /**
22    * interface to be promoted as composite component service
23    * @return
24    */
25   public void  doWork(List<Integer> tasks) {
26     slaves.doWork(tasks);
27     return;
28   }
```

```
29
30   /**
31    * methods require because of the composite (Binding interface)
32    */
33
34   @Override
35   public void bindFc(String clientItfName, Object serverItf)
36       throws NoSuchInterfaceException, IllegalBindingException,
37       IllegalLifeCycleException {
38     if(clientItfName.equals("simple-component-interface")) {
39       slaves = (SimpleComponentMulticastInterface) serverItf;
40     } else {
41       throw new NoSuchInterfaceException(clientItfName);
42     }
43   }
44
45   @Override
46   public String[] listFc() {
47     String[] s =  {"simple-component-interface"};
48     return s;
49   }
50
51   @Override
52   public Object lookupFc(String clientItfName) throws
         NoSuchInterfaceException {
53     if(clientItfName.equals("simple-component-interface"))
54       return slaves;
55     else
56       throw new NoSuchInterfaceException(clientItfName);
57   }
58
59   @Override
60   public void unbindFc(String clientItfName) throws
         NoSuchInterfaceException,
61       IllegalBindingException, IllegalLifeCycleException {
62     if(clientItfName.equals("simple-component-interface"))
63       slaves = null;
64     else
65       throw new NoSuchInterfaceException(clientItfName);
66   }
67
68 }
```

The collector just prints out what it received as parameter on the gathercast port:

Listing 6.40: Collector.java

```
1 package completeOneWay;
2
3 import java.util.List;
4
5 public class Collector implements CollectorInterface {
6
7   @Override
8   public void deliver(List<Integer> res) {
9     java.util.Iterator<Integer> it = res.iterator();
```

```
10     while(it.hasNext()) {
11       System.err.println(":: result time :: "+it.next());
12     }
13     System.err.println(":: empty list left");
14   }
15
16 }
```

In order to denote the correct signature for the gathercast client interface, we also use the following interface:

Listing 6.41: CollectorGathercastInterface.java

```
1 package completeOneWay;
2
3 import java.util.List;
4
5 public interface CollectorGathercastInterface {
6   public void deliver(Integer res);
7
8 }
```

The important files are the `fractal` ones, as usual. In the `Collector` descriptor we must declare the gathercast server port with a `gathercast` cardinality:

Listing 6.42: Collector.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="completeOneWay.adl.Collector">
6
7   <interface
8     name="collector-component-interface"
9     signature="completeOneWay.CollectorInterface"
10    cardinality="gathercast"
11    role="server" />
12
13  <content
14    class="completeOneWay.Collector" />
15
16  <controller
17    desc="primitive" />
18
19  <!-- <virtual-node name="simple-component-node" cardinality="
        single" /> -->
20 </definition>
```

while in the `SimpleComponent` (the worker) descriptor, the cardinality of the corresponding client port shoudl be declared `singleton` (line 16) and the signature should be the modified one (line 9):

Listing 6.43: SimpleComponent.fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
      proactive.dtd">
4
5 <definition name="completeOneWay.adl.SimpleComponent">
6
7   <interface
8     name="simple-component-interface"
9     signature="completeOneWay.SimpleComponentInterface"
10    role="server" />
11
12  <interface
13    name="collector-component-interface"
14    signature="completeOneWay.CollectorGathercastInterface"
15    role="client"
16  cardinality="singleton"/>
17
18  <content
19    class="completeOneWay.SimpleComponent" />
20
21  <controller
22    desc="primitive" />
23
24  <!-- <virtual-node name="simple-component-node" cardinality="
        single" /> -->
25 </definition>
```

The test program at this point could be:

Listing 6.44: Test.java

```
1 package completeOneWay;
2
3 import java.util.ArrayList;
4 import java.util.HashMap;
5 import java.util.Iterator;
6 import java.util.List;
7 import java.util.Map;
8
9 import org.objectweb.fractal.adl.Factory;
10 import org.objectweb.fractal.api.Component;
11 import org.objectweb.fractal.util.Fractal;
12
13 import adles2.SecondComponentInterface;
14
15 public class Test {
16
17   /**
18    * @param args
19    */
20   public static void main(String[] args) {
21     try {
22       // get factory to instantiate the component from ADL
23       Factory f = org.objectweb.proactive.core.component.adl.
             FactoryFactory.getFactory();
```

```
24        // the has table is used to set up the context
25        Map<String, Object> context = new HashMap<String, Object>();
26
27        Component composite = (Component) f.newComponent("
              completeOneWay.adl.Composite", context);
28        System.out.println(":: Composite Component created ...");
29
30        Fractal.getLifeCycleController(composite).startFc();
31        System.out.println(":: Component started ...");
32
33
34        MasterComponentInterface sc = ((MasterComponentInterface)
              composite.getFcInterface("outer-component-interface"));
35
36        List<Integer> params = new ArrayList<Integer>();
37        for(int i=0; i<2; i++)
38          params.add(new Integer(i));
39
40        System.err.print(":: list of params is = <");
41        Iterator<Integer> it = params.iterator();
42        while(it.hasNext())
43          System.err.print(it.next()+":");
44        System.err.println(">");
45
46        sc.doWork(params);
47
48        System.err.println(":: computed results with multicast nodes
              ...");
49
50      } catch (Exception e) {
51        e.printStackTrace();
52        System.exit(0);
53      }
54
55    }
56
57  }
```

Executing the program we'll get an output such as:

Listing 6.45: output.txt

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.completeOneWay._StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
      _StubProActiveInterfaceImpl
6 Generating class : pa.stub.completeOneWay._StubSimpleComponent
7 Generating class : pa.stub.completeOneWay._StubCollector
8 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
9 :: Composite Component created ...
10 :: Component started ...
11 :: list of params is = <0:1:>
12 :: computed results with multicast nodes ...
```

```
13 :: doWork computing 0 at completeOneWay.SimpleComponent@2aed913b got
       task 0:java.lang.Integer
14 :: doWork delivered! (at completeOneWay.SimpleComponent@2aed913b)
15 :: doWork computing 1 at completeOneWay.SimpleComponent@69bcf8f6 got
       task 1:java.lang.Integer
16 :: doWork delivered! (at completeOneWay.SimpleComponent@69bcf8f6)
17 :: result time :: 1
18 :: result time :: 2
19 :: empty list left
```

showing that the two tasks have been actually computed by the two workers. If we use instead a longer list as input, we will get (after changing the mode from `ONE_TO_ONE` to `ROUND_ROBIN` in `SimpleComponentMulticastInterface`):

Listing 6.46: output1.txt

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.completeOneWay._StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
      _StubProActiveInterfaceImpl
6 Generating class : pa.stub.completeOneWay._StubSimpleComponent
7 Generating class : pa.stub.completeOneWay._StubCollector
8 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
9 :: Composite Component created ...
10 :: list of params is = <0:1:2:3:4:5:>
11 :: Component started ...
12 :: computed results with multicast nodes ...
13 :: doWork computing 0 at completeOneWay.SimpleComponent@4c78b613 got
       task 0:java.lang.Integer
14 :: doWork computing 1 at completeOneWay.SimpleComponent@79bd18e4 got
       task 1:java.lang.Integer
15 :: doWork delivered! (at completeOneWay.SimpleComponent@79bd18e4)
16 :: doWork computing 3 at completeOneWay.SimpleComponent@79bd18e4 got
       task 3:java.lang.Integer
17 :: doWork delivered! (at completeOneWay.SimpleComponent@79bd18e4)
18 :: doWork computing 5 at completeOneWay.SimpleComponent@79bd18e4 got
       task 5:java.lang.Integer
19 :: doWork delivered! (at completeOneWay.SimpleComponent@79bd18e4)
20 :: doWork delivered! (at completeOneWay.SimpleComponent@4c78b613)
21 :: doWork computing 2 at completeOneWay.SimpleComponent@4c78b613 got
       task 2:java.lang.Integer
22 :: doWork delivered! (at completeOneWay.SimpleComponent@4c78b613)
23 :: doWork computing 4 at completeOneWay.SimpleComponent@4c78b613 got
       task 4:java.lang.Integer
24 :: doWork delivered! (at completeOneWay.SimpleComponent@4c78b613)
25 :: result time :: 1
26 :: result time :: 2
27 :: empty list left
28 :: result time :: 3
29 :: result time :: 4
30 :: empty list left
31 :: result time :: 5
32 :: result time :: 6
```

```
33 :: empty list left
```

The following output makes more clear what's going on with the gathercast interface:

Listing 6.47: outputSleep.txt

```
1 Generating class : pa.stub.completeOneWay._StubCollector
2 Generating class : pa.stub.org.objectweb.proactive.core.component.
     type._StubComposite
3 :: Composite Component created ...
4 :: Component started ...
5 :: list of params is = <0:1:2:3:4:5:>
6 :: computed results with multicast nodes ...
7 :: doWork computing 0 at completeOneWay.SimpleComponent@3a8905fd got
      task 0:java.lang.Integer
8 :: doWork computing 1 at completeOneWay.SimpleComponent@25e91fa3 got
      task 1:java.lang.Integer
9 :: doWork delivered! (at completeOneWay.SimpleComponent@3a8905fd)
10 :: doWork delivered! (at completeOneWay.SimpleComponent@25e91fa3)
11 :: doWork computing 2 at completeOneWay.SimpleComponent@3a8905fd got
      task 2:java.lang.Integer
12 :: doWork computing 3 at completeOneWay.SimpleComponent@25e91fa3 got
      task 3:java.lang.Integer
13 :: result time :: 1
14 :: result time :: 2
15 :: empty list left
16 :: doWork delivered! (at completeOneWay.SimpleComponent@3a8905fd)
17 :: doWork computing 4 at completeOneWay.SimpleComponent@3a8905fd got
      task 4:java.lang.Integer
18 :: doWork delivered! (at completeOneWay.SimpleComponent@25e91fa3)
19 :: doWork computing 5 at completeOneWay.SimpleComponent@25e91fa3 got
      task 5:java.lang.Integer
20 :: result time :: 3
21 :: result time :: 4
22 :: empty list left
23 :: doWork delivered! (at completeOneWay.SimpleComponent@3a8905fd)
24 :: doWork delivered! (at completeOneWay.SimpleComponent@25e91fa3)
25 :: result time :: 5
26 :: result time :: 6
27 :: empty list left
```

We added a `Thread.sleep(2000)` in the `SingleComponent` `doWork` (the uncommented lines 21–25 in listing 6.38). Therefore each `SingleComponent` waits a bit (2 seconds) before delivering the result to the collector gathercast interface.

Clearly, the gathercast interface on the `Collector` component waits that *all* the items produced by the client gathercast components arrive to the gathercast server before before delivering the result. Therefore, the 6 elements in the input list are processed in 3 chunks (we have only 2 `SimpleComponent` instances). At each chunk, each `SingleComponent` instance produces a result and the `Collector` gathercast interface returns a `List` of results once all the gathercast client components have delivered their results. The overall result is that the `Collector` component delivers three results, one per chunk. This was hap-

pening also in 6.46 but the negligible time spend in `SimpleComponent` computation of `doWork` and the thread scheduling result in an output that seems to indicate the `SimpleComponent` instances first compute all the chunks and then, eventually, the `Collector` outputs the results.

Last but not least, take into account the `Collector` only succeeds delivering when getting one value from *all* the `SimpleComponent` component instances. Therefore, in case the list of integers used is not a multiple of the number of `SimpleComponent` instances, the last, incomplete chunk of results will not be shown by the `Collector.deliver` that will keep staying blocked awaiting for the missing values from the `SimpleComponent` instances that actually did not get anything to compute.

There is another important point to make here. The `Test` program has no more a `System.exit` at the end. As a result, the program does not terminate. If we put back the `System.exit` as in the former versions of the test program, the output is not correct. The reason is that:

- `System.exit` terminates all the thread in the current process
- we used components instantiated in the same JVM of the `main` program
- the `doWork` call in the `Test` program is obviously asynchronous, as usual in ProActive, and in fact the message
  `computed results with multicast nodes ...`
  appears quite early on the output
- therefore the `System.exit` also terminates the running `SimpleComponent` instance (while in `Thread.sleep`) and the `Collector` component *before* they complete their execution.

By the way, this means the code listed in previous Sections relative to the `Test` programs are all "wrong" in that they terminate with the `System.exit`. Actually, the effect we see on the input is the correct one, as the timings involved in component execution are negligible and the Java thread scheduler happens to handle the threads executing components before the one executing the `System.exit`.

In general, the termination of a component composite, once it has been set up and started, requires much more sophisticated techniques. This is obviously due to the fact components are assumed to be somehow "persistent" in the component framework. In order to terminate the components in the correct order, we should pass through the component assembly a kind of "end-of-operation" mark that causes the termination of the component operations (e.g. by calling a proper `terminate` component server interface. Then, the main setting up the composite should query the last component terminating with a a proper blocking server port, and only at that point it can invoke the `System.exit`. This is obviously a good solution for the single JVM, but it can be easily adapted to a distributed component composite.

A very inefficient and naif way[6] of handling termination is the following one:

[6]it is shown here just to evidence some more "typical usage" of GCM components, actually

- we modify the `Collector` to implement two further server ports: one to get the total number of results to be eventually delivered, and the second one to check whether the computation is terminated (the collector got the expected number of results) or not
- we modify the user code (`Test`) in such a way the amount of tasks forwarded to the inner `SimpleComponent` instances is communicated to the `Collector`
- we ask the `Collector` whether the computation is finished before actually shutting down the composite component.

To implement this strategy, we need to change the `Collector` in such a way it implements the `TerminatorInterface`:

Listing 6.48: TerminatorInterface.java

```
1 package completeOneWayTerminating;
2
3 public interface TerminatorInterface {
4
5     public boolean hasTerminated();
6     public void setTaskNo(int t);
7
8 }
```

therefore the code of the collector will look like:

Listing 6.49: Collector.java

```
1 package completeOneWayTerminating;
2
3 import java.util.List;
4
5 public class Collector implements CollectorInterface,
      TerminatorInterface {
6
7   int ntasks = 0;          // to be communicated by master, actually
        ...
8   int i = 0;         // the number of results delivered
9
10
11  @Override
12  public void deliver(List<Integer> res) {
13    System.err.println(":: deliver ENTERED ");
14    java.util.Iterator<Integer> it = res.iterator();
15    while(it.hasNext()) {
16      System.err.println(":: result time :: "+it.next());
17      i++;
18    }
19    System.err.println(":: deliver terminating ");
20  }
21
22  @Override
23  public boolean hasTerminated() {
24    System.err.println(":: hasTerminated entered ...");
```

```
25     if(i==ntasks)
26     return true;
27     else
28     return false;
29   }
30
31   public void setTaskNo(int t) {
32     ntasks = t;
33     return;
34   }
35
36
37 }
```

It is worth pointing out that the `Collector` now implements to interfaces, and the interfaces are exposed in the component descriptor file in two distinct `interface` clauses:

Listing 6.50: Collector. fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
      EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
       proactive.dtd">
4
5 <definition name="completeOneWayTerminating.adl.Collector">
6
7   <interface
8     name="collector-component-interface"
9     signature="completeOneWayTerminating.CollectorInterface"
10    cardinality="gathercast"
11    role="server" />
12
13  <interface
14    name="collector-terminator-interface"
15    signature="completeOneWayTerminating.TerminatorInterface"
16    role="server" />
17
18  <content
19    class="completeOneWayTerminating.Collector" />
20
21  <controller
22    desc="primitive" />
23
24  <!-- <virtual-node name="simple-component-node" cardinality="
        single" /> -->
25 </definition>
```

The `collector-terminator-interface` interface will be exposed at the composite level as a server port, in such a way the client code may invoke the services behind the `hasTerminated` and `setTaskNo` ports. The composite descriptor file, in fact, will expose the `collector-terminator-interface` (lines 12–15, declaration of the interface, and 51–53, binding):

Listing 6.51: Composite. fractal

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//
     EN"
3  "classpath://org/objectweb/proactive/core/component/adl/xml/
      proactive.dtd">
4
5 <definition name="completeOneWayTerminating.adl.Composite">
6
7   <interface
8     name="outer-component-interface"
9     signature="completeOneWayTerminating.MasterComponentInterface"
10    role="server" />
11
12  <interface
13    name="outer-terminator-interface"
14    signature="completeOneWayTerminating.TerminatorInterface"
15    role="server" />
16
17  <component
18    name="master-component"
19    definition="completeOneWayTerminating.adl.MasterComponent" />
20
21  <component
22    name="simple-component-1"
23    definition="completeOneWayTerminating.adl.SimpleComponent" />
24
25  <component
26    name="simple-component-2"
27    definition="completeOneWayTerminating.adl.SimpleComponent" />
28
29  <component
30    name="collector-component"
31    definition="completeOneWayTerminating.adl.Collector" />
32
33  <binding
34    client="master-component.simple-component-interface"
35    server="simple-component-1.simple-component-interface" />
36  <binding
37    client="master-component.simple-component-interface"
38    server="simple-component-2.simple-component-interface" />
39
40  <binding
41    client="simple-component-1.collector-component-interface"
42    server="collector-component.collector-component-interface" />
43  <binding
44    client="simple-component-2.collector-component-interface"
45    server="collector-component.collector-component-interface" />
46
47  <binding
48    client="this.outer-component-interface"
49    server="master-component.master-component-interface" />
50
51  <binding
52    client="this.outer-terminator-interface"
53    server="collector-component.collector-terminator-interface" />
54
```

```
55   <controller desc="composite"/>
56
57 </definition>
```

There are no other changes in the descriptor files. The `Test` code will be changed accordingly to the new interface sported by the `Collector`:

Listing 6.52: Test.java

```
1 package completeOneWayTerminating;
2
3 import java.util.ArrayList;
4 import java.util.HashMap;
5 import java.util.Iterator;
6 import java.util.List;
7 import java.util.Map;
8
9 import org.objectweb.fractal.adl.Factory;
10 import org.objectweb.fractal.api.Component;
11 import org.objectweb.fractal.util.Fractal;
12
13 import adles2.SecondComponentInterface;
14
15 public class Test {
16
17   /**
18    * @param args
19    */
20   public static void main(String[] args) {
21     try {
22       // get factory to instantiate the component from ADL
23       Factory f = org.objectweb.proactive.core.component.adl.
              FactoryFactory.getFactory();
24       // the has table is used to set up the context
25       Map<String, Object> context = new HashMap<String, Object>();
26
27       Component composite = (Component) f.newComponent("
              completeOneWayTerminating.adl.Composite", context);
28       System.out.println(":: Composite Component created ...");
29
30       Fractal.getLifeCycleController(composite).startFc();
31       System.out.println(":: Component started ...");
32
33
34       MasterComponentInterface sc = ((MasterComponentInterface)
              composite.getFcInterface("outer-component-interface"));
35       TerminatorInterface terminator = ((TerminatorInterface)
              composite.getFcInterface("outer-terminator-interface"));
36
37       int taskNo = 3;
38       List<Integer> params = new ArrayList<Integer>();
39       for(int i=0; i<taskNo; i++)
40         params.add(new Integer(i));
41
42       terminator.setTaskNo(taskNo); // tell the collector how many
              results should be awaited
43
```

```
44        System.err.print(":: list of params is = <");
45        Iterator<Integer> it = params.iterator();
46        while(it.hasNext())
47          System.err.print(it.next()+":");
48        System.err.println(">");
49
50        sc.doWork(params);
51
52        // handling termination, the busy wait way (too bad!)
53        System.err.println(":: called computation on inner nodes via
              multicast interface (scatter) ");
54        while(!terminator.hasTerminated()) {
55          try { Thread.sleep(1000); } catch(InterruptedException e) {}
56          System.err.println(":: TEST awaiting termination of
                collector ...");
57        }
58        System.err.println(":: collector terminated\n:: stopping
              component(s) ");
59        // stopping the composite component (and all the inner
              components as a consequence)
60        Fractal.getLifeCycleController(composite).stopFc();
61
62        System.err.println(":: exiting");
63        System.exit(0);
64
65      } catch (Exception e) {
66        e.printStackTrace();
67        System.exit(0);
68      }
69
70    }
71
72  }
```

At line 35, we load the second interface exported (promoted via binding at lines 51–53 in listing 6.51) from the collector. This interface is used at line 42 to communicate to the `Collector` the number of results to be awaited and in the loop at lines 54–57 to busy-wait the termination of the `Collector` component activities[7].

After exiting the loop, we know the `Collector` has terminated its activity and therefore we can stop the component (`stopFc()` call at line 60; this actually stops all the inner components of the composite) and exit (`System.exit` at line 67).

Handling termination the right way

The way we handled termination of the component assembly in the example above works but uses busy wait, which is a technique to be avoided in general, as it wastes CPU resources polling the `Collector` component. Furthermore, the timeout chosen to wait for next polling (line 55 in listing 6.52) influences

[7] the busy-wait technique is deprecated, however. Properly synchronized methods in the `Collector` should be used insted, to block the call to `hasTerminated()` up to the moment the `Collector` has actually received the awaited number of results

the efficiency: a too small value increases useless CPU usage, a too large value impairs reactivity.

A more correct implementation would use a blocking `hasTerminated` method in the collector, in such a way the call to the `hasTerminated` server port is completed only after the `Collector` has finished delivering all the results awaited. This can be accomplished using standard Java programming techniques (synchronized methods plus wait and notify). Unfortunately, ProActive/GCM implements port services in a component as a kind of single threaded activity. Therefore a call to the (blocking) `hasTerminated()` server port on the `Collector` will result in a call to the `wait()` and all the subsequent calls to the `Collector.deliver(...)`, possibily leading to the call of the `notify()` that eventually unblocks the `hasTerminated`, will queued in the component request queue without any chance to be actually executed. In a sense, the server ports of a ProActive/GCM component behave as *mutually exclusive*.

This can be simply verified as follows. We implement a `SyncPlain` class with the aim of implementing a blocking `hasTerminated`:

Listing 6.53: SyncPlain.java

```
1 package completeOneWayTerminatingWait;
2
3 import java.util.concurrent.locks.Condition;
4 import java.util.concurrent.locks.Lock;
5 import java.util.concurrent.locks.ReentrantLock;
6
7 public class SyncPlain {
8
9   private int i = 0;
10  private int limit = 0;
11
12  public SyncPlain(int limit) {
13    this.limit = limit;
14  }
15
16  public synchronized void deliver() {
17    System.out.println(":: deliver locked >> got the "+i+"-th value
        ");
18    i++;
19    if(i==limit) {
20      System.out.println(":. deliver >> signalling  ...");
21      notifyAll();
22    }
23    System.out.println(":: deliver locked >> return");
24    return;
25  }
26
27  public synchronized boolean hasTerminated(int lim) {
28    System.out.println(":: hasTerminated locked >> entered");
29    while(lim != i) {
30      try {
31        System.out.println(":: going to await()");
32        wait();
33        System.out.println("awaited");
```

```
34       } catch (InterruptedException e) {
35         e.printStackTrace();
36       }
37     }
38     System.out.println(":: hasTerminated locked >> return true");
39     return true;
40   }
41 }
```

and then we modify the `Collector` component to use such synchronizer:

Listing 6.54: Collector.java

```
1 package completeOneWayTerminatingWait;
2
3 import java.util.List;
4
5 import org.objectweb.proactive.Body;
6 import org.objectweb.proactive.api.PAActiveObject;
7
8 public class Collector implements CollectorInterface,
      TerminatorInterface {
9
10   int ntasks = 0;          // to be communicated by master, actually
        ...
11   int i = 0;         // the number of results delivered
12   SyncPlain mon = null;      // used to block the hasTerminated
        requester
13
14   public Collector() {
15     // mon = new Sync(ntasks);  // moved below, otherwise
          initialized to 0
16   }
17
18   @Override
19   public void deliver(List<Integer> res) {
20     System.err.println(":: deliver ENTERED ");
21     java.util.Iterator<Integer> it = res.iterator();
22     while(it.hasNext()) {
23       System.err.println(":: result time :: "+it.next());
24       i++;
25       System.out.println("going to call mon.deliver()");
26       mon.deliver();
27       System.out.println("called mon.deliver()");
28     }
29     System.err.println(":: deliver terminating ");
30   }
31
32   @Override
33   public boolean hasTerminated() {
34     System.err.println(":: hasTerminated entered ...");
35     mon.hasTerminated(ntasks);
36     return true;
37   }
38
39   public void setTaskNo(int t) {
40     ntasks = t;
```

```
41      mon = new SyncPlain(ntasks);       // be carefull : this works
            only if the setTaskNo is the first service called !!!!
42      return;
43    }
44
45  }
```

Basically, each time we get a new result (an item of the list, not the list itself) we will issue a `SyncPlain deliver` call (line 26 in listing 6.54. The call, in turn, will issue a `notify` if and only if we have reached the correct number of results (lines 19–22 in listing 6.53). The `hasTerminated` server port method just blocks on the wait hidden in the `SyncPlain hasTerminated` call (line 35 in listing 6.54). However, running this sample code we observe a behaviour such as the following:

Listing 6.55: outputBlock.txt

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.completeOneWayTerminatingWait.
     _StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
     _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
     _StubProActiveInterfaceImpl
6 Generating class : pa.stub.completeOneWayTerminatingWait.
     _StubSimpleComponent
7 Generating class : pa.stub.completeOneWayTerminating.
     _StubSimpleComponent
8 Generating class : pa.stub.completeOneWayTerminatingWait.
     _StubCollector
9 Generating class : pa.stub.org.objectweb.proactive.core.component.
     type._StubComposite
10 :: Composite Component created ...
11 :: Component started ...
12 :: list of params is = <0:1:2:3:>
13 :: called computation on inner nodes via multicast interface (
     scatter)
14 :: hasTerminated entered ...
15 :: hasTerminated locked >> entered
16 :: going to await()
17 :: doWork computing 0 at completeOneWayTerminatingWait.
     SimpleComponent@6b58ba2b got task 0:java.lang.Integer
18 :: doWork computing 1 at completeOneWayTerminating.
     SimpleComponent@2c6ff930 got task 1:java.lang.Integer
19 :: doWork delivered! (at completeOneWayTerminatingWait.
     SimpleComponent@6b58ba2b)
20 :: doWork computing 2 at completeOneWayTerminatingWait.
     SimpleComponent@6b58ba2b got task 2:java.lang.Integer
21 :: doWork delivered! (at completeOneWayTerminating.
     SimpleComponent@2c6ff930)
22 :: doWork computing 3 at completeOneWayTerminating.
     SimpleComponent@2c6ff930 got task 3:java.lang.Integer
23 :: doWork delivered! (at completeOneWayTerminatingWait.
     SimpleComponent@6b58ba2b)
```

```
24 :: doWork delivered! (at completeOneWayTerminating.
      SimpleComponent@2c6ff930)
```

At line 16, we see the `wait` is going to be called. This actually makes the `Collector` component busy serving the `hasTerminated` port. The subsequent issues of `deliver` calls by the `SimpleComponent` instances (lines 17–23) result in enqueuing requests in the `Collector` component unique request queue that will never be served as the component is serving the previous `hasTerminated` request. In fact, the execution blocks at this point and you'll see no further output.

To solve the problem, underlying ProActive (non GCM) mechanisms have to be used. In particular, ProActive/GCM components are implemented and Active Objects, the main ProActive abstraction. Active Objects support *immediated services* i.e. requests[8] not flowing through the standard, unique, request queue, that are immediately served. Such a feature can be activated implementing the `ComponentInitActive` interface. The interface requires the implementation of a `public void initComponentActivity(Body arg0)` method where a call to the static method
`PAActiveObject.setImmediateService(String methodName)`
can be placed. The `initComponentActivity` method is executed when the component is started. The call to `setImmediateService` allows request directed to the named string to be executed immediately, without actually being enqueued in the unique component request queue.

With this further knowledge, we can modify our `Collector` code as follows:

Listing 6.56: Collector.java

```
1 package completeOneWayTerminatingWait;
2
3 import java.util.List;
4
5 import org.objectweb.proactive.Body;
6 import org.objectweb.proactive.api.PAActiveObject;
7 import org.objectweb.proactive.core.component.body.
      ComponentInitActive;
8
9 public class Collector implements CollectorInterface,
      TerminatorInterface, ComponentInitActive {
10
11   int ntasks = 0;         // to be communicated by master, actually
        ...
12   int i = 0;        // the number of results delivered
13   SyncPlain mon = null;     // used to block the hasTerminated
        requester
14
15   public Collector() {
16     // mon = new Sync(ntasks);  // moved below, otherwise
          initialized to 0
17   }
18
```

[8]method requests and, as a consequence, server port invocations

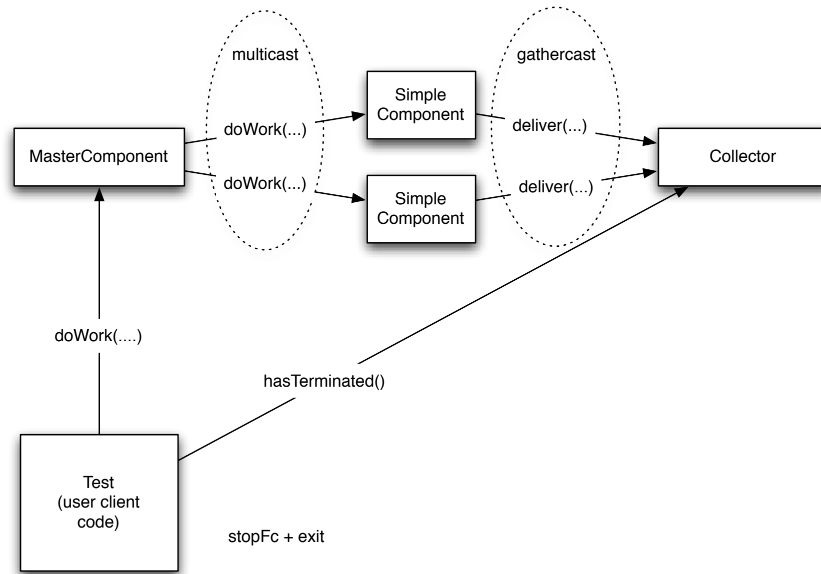

Figure 6.4: Component assembly with major interactions

```
19    @Override
20    public void deliver(List<Integer> res) {
21      System.err.println(":: deliver ENTERED ");
22      java.util.Iterator<Integer> it = res.iterator();
23      while(it.hasNext()) {
24        System.err.println(":: result time :: "+it.next());
25        i++;
26        System.out.println("going to call mon.deliver()");
27        mon.deliver();
28        System.out.println("called mon.deliver()");
29      }
30      System.err.println(":: deliver terminating ");
31    }
32
33    @Override
34    public boolean hasTerminated() {
35      System.err.println(":: hasTerminated entered ...");
36      mon.hasTerminated(ntasks);
37      return true;
38    }
39
40    public void setTaskNo(int t) {
41      ntasks = t;
42      mon = new SyncPlain(ntasks);      // be carefull : this works
          only if the setTaskNo is the first service called !!!!
43      return;
44    }
45
```

```
46   @Override
47   public void initComponentActivity(Body arg0) {
48     // TODO Auto-generated method stub
49     PAActiveObject.setImmediateService("hasTerminated");
50   }
51
52
53 }
```

At line 9, we implement `ComponentInitActive` interface in addition to the other two, application dependent interfaces. Then, from line 46 to 50, we implement the `initComponentActivity` method inserting in its body the call to the `setImmediateService` of the `hasTerminated` method. This frees the component input queue of the blocking request to `hasTerminated` and as a consequence, the queue is free to serve the `deliver` server port calls that eventually will lead to the `notify` call that unblocks the `hasTerminated` call. Figure 6.4 shows the structure of our code, at this stage, along with the major interactions among components.

If we run this modified version of our code, we will get an output such as:

Listing 6.57: output.txt

```
1  --> This ClassFileServer is listening on port 2026
2 Created a new registry on port 1099
3 Generating class : pa.stub.completeOneWayTerminatingWait.
      _StubMasterComponent
4 Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
      _StubJMXNotificationListener
5 Generating class : pa.stub.org.objectweb.proactive.core.component.
      _StubProActiveInterfaceImpl
6 Generating class : pa.stub.completeOneWayTerminatingWait.
      _StubSimpleComponent
7 Generating class : pa.stub.completeOneWayTerminating.
      _StubSimpleComponent
8 Generating class : pa.stub.completeOneWayTerminatingWait.
      _StubCollector
9 Generating class : pa.stub.org.objectweb.proactive.core.component.
      type._StubComposite
10 :: Composite Component created ...
11 :: Component started ...
12 :: list of params is = <0:1:2:3:>
13 :: called computation on inner nodes via multicast interface (
      scatter)
14 :: hasTerminated entered ...
15 :: hasTerminated locked >> entered
16 :: going to await()
17 :: doWork computing 0 at completeOneWayTerminatingWait.
      SimpleComponent@13c0210e got task 0:java.lang.Integer
18 :: doWork computing 1 at completeOneWayTerminating.
      SimpleComponent@43781cf8 got task 1:java.lang.Integer
19 :: doWork delivered! (at completeOneWayTerminatingWait.
      SimpleComponent@13c0210e)
20 :: doWork computing 2 at completeOneWayTerminatingWait.
      SimpleComponent@13c0210e got task 2:java.lang.Integer
```

```
21 :: doWork delivered! (at completeOneWayTerminating.
      SimpleComponent@43781cf8)
22 :: doWork computing 3 at completeOneWayTerminating.
      SimpleComponent@43781cf8 got task 3:java.lang.Integer
23 :: deliver ENTERED
24 :: result time :: 1
25 going to call mon.deliver()
26 :: result time :: 2
27 :: deliver terminating
28 :: deliver locked >> got the 0-th value
29 :: deliver locked >> return
30 called mon.deliver()
31 going to call mon.deliver()
32 :: deliver locked >> got the 1-th value
33 :: deliver locked >> return
34 called mon.deliver()
35 :: doWork delivered! (at completeOneWayTerminatingWait.
      SimpleComponent@13c0210e)
36 :: doWork delivered! (at completeOneWayTerminating.
      SimpleComponent@43781cf8)
37 going to call mon.deliver()
38 :: deliver ENTERED
39 :: result time :: 3
40 :: result time :: 4
41 :: deliver terminating
42 :: deliver locked >> got the 2-th value
43 :: deliver locked >> return
44 called mon.deliver()
45 going to call mon.deliver()
46 :: deliver locked >> got the 3-th value
47 :. deliver >> signalling  ...
48 :: deliver locked >> return
49 called mon.deliver()
50 awaited
51 :: hasTerminated locked >> return true
52 :: collector terminated
53 :: stopping component(s)
54 :: exiting
```

As you can see, the `wait` is called (line 16) as a consequence of the call to the `hasTerminated` server port (lines 14–15). Then `SimpleComponent` instances compute tasks 0 and 1 (lines 17–18), deliver the corresponding results (lines 19 and 21) to the gathercast `Collector` port and start computing tasks 2 and 3 (lines 20 and 22). The collector manages the delivered values for the first list (results of tasks 0 and 1) at lines 23–34. The the `SimpleComponent` instances deliver the results of tasks 2 and 3 (four tasks have been generated in this run) at lines 35–36 and eventually, the `Collector` component processes this calls to the gathercast `deliver` port (lines 38–49). While delivering the fourth result, the `notify` call is issued and this resumes execution of the `hasTerminated` server port (line 50–51) that in turn allows the `Test` code to invoke the `stopFc` and to terminate (lines 53–54).

6.1.9 Classpath and Java 1.6

Java 1.6 introduced facilities to work with the classpath. In particular, the class path can be expressed as a directory followed by an asterisk (*). In this case *all* the `.jar` files in the directory will be included in the class path.

This is useful in at least two different contexts:

- when using the compiler or the interpreter from the command line. As an example, one can (using a `bash`):

  ```
  [marcod@u5]$ export CLASSPATH=.:/$HOME/ProActive/dist/lib/\*
  [marcod@u5]$ javac vnes0/*java
  [marcod@u5]$
  ```

  i.e. you can export in the class path the directory where the ProActive `.jar` files are located followed by the asterisk wild card and then successfully compile the `.java` using ProActive.

- when using ProActive descriptors, you can subsume all the lines hosting a single `.jar` to be inserted in the class path (e.g. the lines 60 to 71 of the Listing 6.23) with an XML code such as:

  ```
  <classpath>
    <absolutePath value="${PROACTIVE_HOME}/dist/lib/\*"/>
  </classpath>
  ```

  Be careful, the asterisk should be quoted with the backslash to prevent expansion when the `-classpath` command is issued on the remote shell.

6.1.10 SCA/Tuscany

Installation

In order to use SCA/Tuscany, download the latest version of Tuscany from the Tuscany web site at `http://tuscany.apache.org/`. Here we assume to use the version 1.4 of the Tuscany/SCA environment, the one available when this notes have been prepared.

Then, un-compact the `zip` or `tgz` file downloaded, and you will get a directory named `tuscany-sca-1.4`. This directory will host in particular two sub-directories: `lib` and `samples`. The `samples` one contains the simple examples discussed in the Tuscany/SCA web pages and tutorials. The `lib` directory contains all the jar files that have to be included in the classpath to compile and run SCA/Tuscany programs.

In Eclipse, as suggested on the documentation on the Tuscany web page, you can create a Library in the project preferences, including all the jar files in the `lib` sub-directory of the SCA/Tuscany distribution.

Sample usage: single component

In this Section, we show how to define a single component within SCA/Tuscany and to instantiate and use the component. The component is the usual `SimpleComponent` we used in the ProActive Section. It is defined as follows:

Listing 6.58: SimpleComponent.java

```
1 package simpleComponent;
2
3 public class SimpleComponent implements SimpleComponentInterface {
4
5   @Override
6   public Integer doWork(Integer task) {
7     int i = task.intValue();
8     System.err.println(":s:: Computing task "+i);
9     return new Integer(++i);
10   }
11
12 }
```

which makes no difference with the `SimpleComponent` used in ProActive. There is one small difference in the `SimpleComponentInterface.java` however, as the component (i.e. its interface) should be annoted as `@Remotable`. This is used to realize that the corresponding implementation code will be used to implement a component.

Listing 6.59: SimpleComponentInterface.java

```
1 package simpleComponent;
2
3 import org.osoa.sca.annotations.Remotable;
4
5 @Remotable
6 public interface SimpleComponentInterface {
7   public Integer doWork(Integer task);
8 }
```

After providing the code implementing the component and describing its interface, we have to provide a component descriptor in a `.composite` file. This simple component is described through the `SimpleComponent.composite` descriptor:

Listing 6.60: SimpleComponent.composite

```
1 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
2            name="SimpleServiceComponent">
3
4     <component name="SimpleServiceComponent">
5   <implementation.java class="simpleComponent.SimpleComponent"/>
6     </component>
7
8 </composite>
```

The descritor simply states the name of the component (at line 4, `SimpleServiceComponent`) and the implementation through a Java class (at line 5). It is a `composite` tag, although in this case the resulting component has no composition inside, it is a single component.

In order to use the component, the simplest way is to create a SCA domain and load the descriptor. The following main show how this can be done:

Listing 6.61: Main.java

```
1 package simpleComponent;
2
3 import org.apache.tuscany.sca.host.embedded.SCADomain;
4
5 public class Main {
6
7   public static void main(String[] args) {
8     System.err.println(":: starting ... ");
9     SCADomain scaDomain = SCADomain.newInstance("SimpleComponent.
          composite");
10    System.err.println(":: domain activated");
11    SimpleComponentInterface simpleComp = scaDomain.getService(
          SimpleComponentInterface.class, "SimpleServiceComponent");
12    System.err.println(":: got simpleComp");
13    Integer x = simpleComp.doWork(new Integer(5));
14    System.err.println(":: terminating ... ");
15  }
16
17 }
```

At line 8, a domain is created using a static method of the `SCADomain` class. The static method is passed the descriptor of the component. The component descriptor (this is a filename, actually) should be placed in the root level of `CLASSPATH`. In our case, we used a package `singleComponent` and the `SimpleComponent.composite` must be located in the directory hosting the `singleComponent` package sub-directory hosting, in turn, the `simpleComponent.SingleComponent` and `singleComponent.Main` class files. At line 11, we get the reference to the `SimpleComponentInterface`[9] by using a proper method in the domain object and passing it the class implement the interface of the component and the name used in the `composite` for the component. Then, at line 13, we can use the component as a normal java object method call.

When we run the `Main` class, we get the following output:

Listing 6.62: output.txt

```
1 :: starting ...
2 Apr 26, 2009 4:29:28 PM org.apache.tuscany.sca.node.impl.NodeImpl <
     init>
3 INFO: Creating node: SimpleComponent.composite
4 Apr 26, 2009 4:29:29 PM org.apache.tuscany.sca.node.impl.NodeImpl
     configureNode
```

[9] be careful, by indicating a reference to a `SimpleComponent` you'll get a run time exception here

```
5 INFO: Loading contribution: file:/Users/marcodanelutto/Documents/
     Ricerca/Muskel/Tuscany-SCA/bin/
6 Apr 26, 2009 4:29:29 PM org.apache.tuscany.sca.assembly.xml.
     CompositeProcessor
7 WARNING: No namespace found: Composite = SCAsample1
8 Apr 26, 2009 4:29:29 PM org.apache.tuscany.sca.assembly.xml.
     CompositeProcessor
9 WARNING: No namespace found: Composite = SimpleServiceComponent
10 Apr 26, 2009 4:29:30 PM org.apache.tuscany.sca.node.impl.NodeImpl
     configureNode
11 INFO: Loading composite: file:/Users/marcodanelutto/Documents/
     Ricerca/Muskel/Tuscany-SCA/bin/SimpleComponent.composite
12 Apr 26, 2009 4:29:30 PM org.apache.tuscany.sca.assembly.xml.
     CompositeProcessor
13 WARNING: No namespace found: Composite = SimpleServiceComponent
14 Apr 26, 2009 4:29:30 PM org.apache.tuscany.sca.node.impl.NodeImpl
     start
15 INFO: Starting node: SimpleComponent.composite
16 :: domain activated
17 :: got simpleComp
18 :s:: Computing task 5
19 :: terminating ...
```

At line 3, we see the composite descriptor is loaded and, as a consequence, the `SimpleComponent` is started at line 15. Then the component is used and the result of its call is shown at line 18. As usual, we use lines starting with `::` to denote our diagnostic messages, distinguished from the ones provided by the implementation.

Sample usage: composite component

Here we discuss how a simple composite component can be built out of two simple component. In particular, as we did for ProActive/GCM, we'll show what are the implications of using a service provide by a component within another component. We will use the same `SimpleComponent` discussed above to implement a `SecondComponent` as already shown in the ProActive section.

The `SimpleComponent.java` and `SimpleComponentInterface.java` files are the same of the ones discussed in the previous section. The `SecondComponent` that eventually will use the `SimpleComponent` is implemented through the following `SecondComponentInterface.java` and `SecondComponent.java` files.

Listing 6.63: SecondComponent.java

```
1 package compositeComponent;
2
3 import org.osoa.sca.annotations.Reference;
4
5 public class SecondComponent implements SecondComponentInterface {
6
7   private SimpleComponentInterface c1;
8
9   @Reference
```

```
10   public void setC1(SimpleComponentInterface x) {
11     System.err.println(":: adding SimpleComponent reference in
           SecondComponent "+x);
12     c1 = x;
13   }
14
15   public Integer compute(Integer task) {
16     System.err.println(":: compute called with
           simpleServiceComponent="+c1);
17     Integer res = c1.doWork(task);
18     int i = res.intValue();
19     return new Integer(2*i);
20   }
21
22 }
```

Listing 6.64: SecondComponentInterface.java

```
1 package compositeComponent;
2
3 import org.osoa.sca.annotations.Remotable;
4
5 @Remotable
6 public interface SecondComponentInterface {
7   public Integer compute(Integer task);
8 }
```

The `SecondComponent` *uses* a `SimpleComponent`. Therefore, in the `SecondComponent.java` implementation file, at line 7, we define a reference to a `SimpleComponentInterface`[10] object. The reference will be filled up when consulting the `Composite.composite` descriptor file that will mention the reference (i.e. the use interface of `SecondComponent`). The `SecondComponent` implementation should therefore provide a `@Reference` annotated method to set up the reference during the component assembly instantiation. This is what is provided in our case in lines 9 to 13 of the `SecondComponent` implementation.

It is worth pointing out that in ProActive/GCM, the presence of `use` interface requires the setup of an explicit `BindingController`, while in case of SCA, only a setter method is needed for the reference field.

The composite component descriptor file can then be provided as follows:

Listing 6.65: Composite.composite

```
1 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
2            name="SCAsample1">
3
4     <component name="SecondComponent">
5   <implementation.java class="compositeComponent.SecondComponent"/>
6         <reference name="c1" target="FirstComponent" />
7     </component>
8
9     <component name="FirstComponent">
```

[10] as before, be careful to use the interface here, not the implementation class, otherwise you'll get errors

```
10   <implementation.java class="compositeComponent.SimpleComponent"/>
11     </component>
12
13 </composite>
```

At lines 9–11 we defined the `SimpleComponent` named `FirstComponent` as shown in the previous Section. At lines 4–7 we describe the first component, the one `using` the `FirstComponent`. Apart from the component name and implementation class details, we have a `reference` tag naming the *exact* name of the instance variable used to referee (to have a reference to) the used component in the implementation of `SecondComponent` and a target, represented by the name of the component that will be eventually bound to the reference during composite component assembly instantiation.

Sample usage: deploymnet

6.2 Skeleton programming environments

6.2.1 Muskel

6.2.2 BS/GCM

6.2.3 SkeTo

6.3 Use cases

6.3.1 Image processing

6.3.2 Game of life

6.3.3 Data center