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

```
package adles2;

public interface SimpleComponentInterface {

  public Integer doWork(Integer task);
}
```

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

Listing 6.2: SimpleComponent.java

```java
package adles2;

public class SimpleComponent implements SimpleComponentInterface {

  public Integer doWork(Integer task) {
    int iv = task.intValue();
    System.err.println("--- doWork computing "+iv+" got task "+task
        +":"+task.getClass().getName());
    Integer r = new Integer(++iv);
    return r;
  }

}
```

Eventually the component descriptor can be given as follows:

Listing 6.3: SimpleComponent.fractal

```xml
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

```java
package adles1;

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

import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.util.Fractal;

public class Main {

  public static void main(String [] args) {

    try {
      // get factory to instantiate the component from ADL
      Factory f = org.objectweb.proactive.core.component.adl.
          FactoryFactory.getFactory();
      // the has table is used to set up the context
          Map<String, Object> context = new HashMap<String, Object
              >();

          Component simpleComponent = (Component) f.newComponent("
              adles1.SimpleComponent", context);
          System.out.println(":: Component created ...");

          Fractal.getLifeCycleController(simpleComponent).startFc();
      System.out.println(":: Component started ...");


      SimpleComponentInterface sc = ((SimpleComponentInterface)
          simpleComponent.getFcInterface("simple-comp-interface"));
      Object n = sc.doWork(new Integer(5));
      System.out.println(":: Object coming back of type : "+n.
          getClass().getName());

      System.out.println(":: Computed sc.doWork(5)="+n.toString());


          Fractal.getLifeCycleController(simpleComponent).stopFc();
      System.out.println(":: Component stopped ...");

      System.exit(0);

    } catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }

  }

}
```

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

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

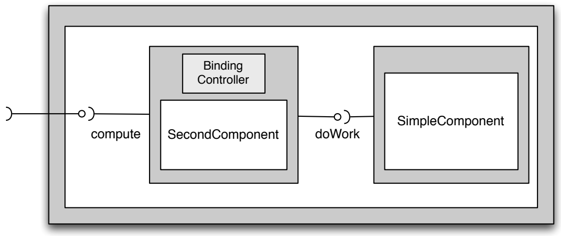

Figure 6.1: Simple component assembly

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

```java
package adles2;

public interface SecondComponentInterface {

  public Integer compute(Integer task);
}
```

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

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

<definition name="adles2.SecondComponent">

  <interface
    name="second-comp-interface"
    signature="adles2.SecondComponentInterface"
    role="server" />

  <interface
    name="simple-comp-interface"
    signature="adles2.SimpleComponentInterface"
    role="client" />

  <content
```

```
    class="adles2.SecondComponent" />

</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

```
package adles2;

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

public class SecondComponent implements SecondComponentInterface,
    BindingController {

  SimpleComponentInterface otherComponent;

  public SecondComponent() {
    // empty
    return;
  }

  @Override
  public Integer compute(Integer task) {
    // call other component
    Integer temp = otherComponent.doWork(task);
    System.err.println("--- called SimpleComponent, got "+temp+"
        type "+temp.getClass().getName());
    // post process result
    int i = temp.intValue();
    // return new result
    return new Integer(i*2);
  }

  @Override
  public void bindFc(String arg0, Object arg1)
      throws NoSuchInterfaceException, IllegalBindingException,
      IllegalLifeCycleException {

    if(arg0.equals("simple-comp-interface")) {
      otherComponent = (SimpleComponentInterface) arg1;
    }
  }
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

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

```

<definition name="adles2.Composite">

  <interface
    name="outer-comp-interface"
    signature="adles2.SecondComponentInterface"
    role="server" />

  <component
    name="second-component"
    definition="adles2.SecondComponent" />

  <component
    name="first-component"
    definition="adles2.SimpleComponent" />

  <binding
    client="second-component.simple-comp-interface"
    server="first-component.simple-comp-interface" />

  <binding
    client="this.outer-comp-interface"
    server="second-component.second-comp-interface" />

</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

```java
package adles2;

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

import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.util.Fractal;

public class Main {

  public static void main(String [] args) {

    try {
      // get factory to instantiate the component from ADL
      Factory f = org.objectweb.proactive.core.component.adl.
          FactoryFactory.getFactory();
      // the has table is used to set up the context
          Map<String, Object> context = new HashMap<String, Object
              >();

          Component composite = (Component) f.newComponent("adles2.
              Composite", context);
          System.out.println(":: Composite Component created ...");

          Fractal.getLifeCycleController(composite).startFc();
      System.out.println(":: Component started ...");


      SecondComponentInterface sc = ((SecondComponentInterface)
          composite.getFcInterface("outer-comp-interface"));
      Object n = sc.compute(new Integer(5));
      System.out.println(":: Object coming back of type : "+n.
          getClass().getName());

      System.out.println(":: Computed sc.compute(5)="+n.toString());
//
//
//          Fractal.getLifeCycleController(simpleComponent).stopFc()
    ;
//      System.out.println(":: Component stopped ...");

      System.exit(0);

    } catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }

  }

}
```

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

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

<definition name="vnes1.SimpleComponent">

  <interface
    name="simple-comp-interface"
    signature="vnes1.SimpleComponentInterface"
    role="server" />

  <content
    class="vnes1.SimpleComponent" />

  <controller
    desc="primitive" />

  <virtual-node name="simple-component-node" cardinality="single" />
</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

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3
 http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/deployment.xsd">

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/Users/marcodanelutto/Documents/Didattica/CCCP/Strumenti/ProActive/ProActive-4.0.2" />
    <descriptorVariable name="JAVA_HOME" value="/usr/bin" />

  </variables>

  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="second-component-node"/>
      <virtualNode name="simple-component-node"/>
    </virtualNodesDefinition>
  </componentDefinition>

  <deployment>

    <mapping>
      <map virtualNode="second-component-node">
         <jvmSet><currentJVM/></jvmSet>
      </map>
      <map virtualNode="simple-component-node">
         <jvmSet><currentJVM/></jvmSet>
      </map>
    </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

```java
package vnes1;

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

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

import vnes1.SecondComponentInterface;

public class Test {

  /**
   * @param args
   */
  public static void main(String[] args) {
    try {
      // get the Factory
      Factory f = (Factory) org.objectweb.proactive.core.component.adl.FactoryFactory.getFactory();
      // create the context
      Map<String, Object> context = new HashMap<String, Object>();
      // find the ProActiveDescriptor (this is the one hosting the virtual node definition and mapping
      ProActiveDescriptor deploymentDescriptor =
        PADeployment.getProactiveDescriptor("file:///Users/marcodanelutto/Documents/Ricerca/Muskel/ProActiveCCP/src/vnes1/vnes1descr.xml");
      // add descriptor to the context
      context.put("deployment-descriptor", deploymentDescriptor);
      // activate mappings
      deploymentDescriptor.activateMappings();

//      Component first = (Component) f.newComponent("vnes1.SimpleComponent", context);
      // now create components
      Component composite = (Component) f.newComponent("vnes1.Composite", context);
          System.out.println(":: Composite Component created ...");

          Fractal.getLifeCycleController(composite).startFc();
      System.out.println(":: Component started ...");


      SecondComponentInterface sc = ((SecondComponentInterface) composite.getFcInterface("outer-comp-interface"));
      Object n = sc.compute(new Integer(5));
      System.out.println(":: Object coming back of type : "+n.getClass().getName());

      System.out.println(":: Computed sc.compute(5)="+n.toString());
//
//
//          Fractal.getLifeCycleController(simpleComponent).stopFc();
//      System.out.println(":: Component stopped ...");

      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

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3
 http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
     deployment.xsd">

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/Users/
        marcodanelutto/Documents/Didattica/CCCP/Strumenti/ProActive/
        ProActive-4.0.2" />
    <descriptorVariable name="JAVA_HOME" value="/usr/bin" />

  </variables>

  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="second-component-node"/>
      <virtualNode name="simple-component-node"/>
    </virtualNodesDefinition>
  </componentDefinition>

  <deployment>

```

[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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3
 http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
     deployment.xsd">

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/home/marcod/
        ProActive-4.0.2" />
    <descriptorVariable name="JAVA_HOME" value="/home/marcod/jdk/bin
        " />

  </variables>

  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="simple-component-node"/>
    </virtualNodesDefinition>
  </componentDefinition>

  <deployment>

    <mapping>
      <map virtualNode="simple-component-node">
  <jvmSet> <vmName value="jvm-2"/> </jvmSet>
      </map>
    </mapping>

    <jvms>
      <jvm name="jvm-2"> <creation> <processReference refid="process
          -distributed-2" /> </creation> </jvm>
    </jvms>

  </deployment>

  <infrastructure>
    <processes>

      <processDefinition id="process-distributed-2">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess
              "
          username="marcod" hostname="u12">
          <processReference refid="process-remote"/>
        </sshProcess>
      </processDefinition>

      <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

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

<definition name="vnes0.SimpleComponent">

  <interface
    name="simple-comp-interface"
    signature="vnes0.SimpleComponentInterface"
    role="server" />

  <content
    class="vnes0.SimpleComponent" />

  <controller
    desc="primitive" />

  <virtual-node name="simple-component-node" cardinality="single" />
</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

```java
package vnes0;

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

import org.objectweb.fractal.adl.Factory;
import org.objectweb.fractal.api.Component;
import org.objectweb.fractal.util.Fractal;
import org.objectweb.proactive.api.PADeployment;
import org.objectweb.proactive.core.descriptor.data.
    ProActiveDescriptor;

public class Main {

  public static void main(String [] args) {

    try {
      // get factory to instantiate the component from ADL
      Factory f = org.objectweb.proactive.core.component.adl.
          FactoryFactory.getFactory();
      // the has table is used to set up the context
          Map<String, Object> context = new HashMap<String, Object
              >();

      ProActiveDescriptor deploymentDescriptor =
        PADeployment.getProactiveDescriptor("vnes0/distributed.xml")
            ;
      // add descriptor to the context
      context.put("deployment-descriptor", deploymentDescriptor);
      // activate mappings
      deploymentDescriptor.activateMappings();
      System.err.println("mapping activated\nnow starting component
          deploymnet");

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

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

```
now starting component deploymnet
**** Starting jvm on 10.0.2.12
 --> This ClassFileServer is listening on port 2030
Detected an existing RMI Registry on port 1099
**** Mapping VirtualNode simple-component-node with Node: rmissh
     ://10.0.2.12:1099/simple-component-node122318164 done
Generating class : pa.stub.vnes0._StubSimpleComponent
Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
     _StubJMXNotificationListener
:: Component created ...
:: Component started ...
Generating class : pa.stub.java.lang._StubObject
--- doWork computing 5 got task 5:java.lang.Integer
:: Object coming back of type : pa.stub.java.lang._StubObject
:: Computed sc.doWork(5)=6
:: Component stopped ...
[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

```java
package vnes0;

public class SimpleComponent implements SimpleComponentInterface {

  @Override
  public Object doWork(Object task) {
                try {
                        String localMachine1 = java.net.InetAddress.
                            getLocalHost().getCanonicalHostName();
                        System.err.println(":: Hostname of local
                            machine: " + localMachine1);
                } catch (Exception e) {
                        e.printStackTrace();
                }

    Integer i = (Integer) task;
    int iv = i.intValue();
    System.err.println("--- doWork computing "+iv+" got task "+task
        +":"+task.getClass().getName());
    Integer r = new Integer(++iv);
    return r;
  }

}
```

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

Listing 6.22: output1

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

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3
 http://www-sop.inria.fr/oasis/ProActive/schemas/deployment/3.3/
     deployment.xsd">

  <variables>
    <descriptorVariable name="REMOTE_PROACTIVE_HOME" value="/home/
        marcod/ProActive-4.0.2" />
    <descriptorVariable name="REMOTE_JAVA_HOME" value="/home/marcod/
        jdk/bin" />

  </variables>

  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="second-component-node"/>
      <virtualNode name="simple-component-node"/>
    </virtualNodesDefinition>
  </componentDefinition>

  <deployment>

    <mapping>
      <map virtualNode="second-component-node">
  <jvmSet> <vmName value="jvm-1"/> </jvmSet>
      </map>
      <map virtualNode="simple-component-node">
  <jvmSet> <vmName value="jvm-2"/> </jvmSet>
      </map>
    </mapping>

```

[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.

```
  <jvms>
    <jvm name="jvm-1"> <creation> <processReference refid="process
        -distributed-1" /> </creation> </jvm>
    <jvm name="jvm-2"> <creation> <processReference refid="process
        -distributed-2" /> </creation> </jvm>
  </jvms>

</deployment>

<infrastructure>
  <processes>

    <processDefinition id="process-distributed-1">
      <sshProcess class="org.objectweb.proactive.core.process.ssh.
          SSHProcess"
        username="marcod" hostname="u12">
        <processReference refid="process-remote"/>
      </sshProcess>
    </processDefinition>

    <processDefinition id="process-distributed-2">
      <sshProcess
        class="org.objectweb.proactive.core.process.ssh.SSHProcess
            "
        username="marcod"
        hostname="u13">
        <processReference refid="process-remote"/>
      </sshProcess>
    </processDefinition>

    <processDefinition id="process-remote">
      <jvmProcess class="org.objectweb.proactive.core.process.
          JVMNodeProcess">
        <classpath>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              ProActive.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              fractal.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              asm-2.2.1.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              bouncycastle.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              dtdparser.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              fractal-adl.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              javassist.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              ganymed-ssh2-build210.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              junit-4.4.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              log4j.jar"/>
          <absolutePath value="${REMOTE_PROACTIVE_HOME}/dist/lib/
              ow_deployment_scheduling.jar"/>
          <absolutePath value="${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

```
[marcod@u5 ~/cccp]$ java -Dfractal.provider=org.objectweb.proactive.
    core.component.Fractive -Djava.security.policy=./all.permissions
     vnes1.Test
:: GOT FACTORY
:: CREATED CONTEXT MAP
 --> This ClassFileServer is listening on port 2026
Created a new registry on port 1099
************* Reading deployment descriptor: file:///home/marcod/
    cccp/vnes1/pianosa.xml *******************
created VirtualNode name=second-component-node
created VirtualNode name=simple-component-node
:: GOT PROACTIVE DESCRIPTOR
:: PUT DESCRIPTOR IN CONTEXT
Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
    _StubJMXNotificationListener
:: ACTIVATED MAPPINGS
mapping activated
now starting component deploymnet
**** Starting jvm on 10.0.2.13
**** Starting jvm on 10.0.2.12
 --> This ClassFileServer is listening on port 2026
Created a new registry on port 1099
 --> This ClassFileServer is listening on port 2026
Created a new registry on port 1099
**** Mapping VirtualNode simple-component-node with Node: rmissh
    ://10.0.2.13:1099/simple-component-node617019055 done
**** Mapping VirtualNode second-component-node with Node: rmissh
    ://10.0.2.12:1099/second-component-node860041102 done
Generating class : pa.stub.vnes1._StubSecondComponent
Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
    _StubJMXNotificationListener
Generating class : pa.stub.vnes1._StubSimpleComponent
Generating class : pa.stub.org.objectweb.proactive.core.jmx.util.
    _StubJMXNotificationListener
Generating class : pa.stub.org.objectweb.proactive.core.component.
    type._StubComposite
:: Composite Component created ...
:: Component started ...
:: SecondComponent :: Hostname of local machine: u12
:: SimpleComponent :: Hostname of local machine: u13
--- doWork computing 5 got task 5:java.lang.Integer
:: hostname: i386
--- called SimpleComponent, got 6 type java.lang.Integer
:: Object coming back of type : java.lang.Integer
:: Computed sc.compute(5)=12
Press <enter> to continue

```

```
node simple-component-node617019055 is being killed, terminating
    body -338392b1-1211190e585--7fee--3e84b5785f29789d--338392b1
    -1211190e585--8000
node Node1366617550 is being killed, terminating body -338392b1
    -1211190e585--7fc4--3e84b5785f29789d--338392b1-1211190e585--8000
terminating Runtime //10.0.2.13/PA_JVM1837501017
Process finished Thread=ERR -> ssh -l marcod u13  -
unable to contact remote object at rmissh://10.0.2.13:1099/
    PA_JVM1837501017 when calling killRT
 Virtual Machine 3e84b5785f29789d:-338392b1:1211190e585:-8000 on
     host 10.0.2.13 terminated.
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
Process finished Thread=IN -> ssh -l marcod u13  -
node second-component-node860041102 is being killed, terminating
    body 1d9792f6-12111878338--7fee--a9ef8807ca30c7ef-1d9792f6
    -12111878338--8000
node Node272236384 is being killed, terminating body 1d9792f6
    -12111878338--7fc4--a9ef8807ca30c7ef-1d9792f6-12111878338--8000
terminating Runtime //10.0.2.12/PA_JVM1723678229
unable to contact remote object at rmissh://10.0.2.12:1099/
    PA_JVM1723678229 when calling killRT
 Virtual Machine a9ef8807ca30c7ef:1d9792f6:12111878338:-8000 on host
      10.0.2.12 terminated.
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
Process finished Thread=IN -> ssh -l marcod u12  -
Process finished Thread=ERR -> ssh -l marcod u12  -
[marcod@u5 ~/cccp]$
```

6.1.6 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.

6.1.7 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 6.25: SimpleComponent.java

```java
package simpleComponent;

public class SimpleComponent implements SimpleComponentInterface {

  @Override
  public Integer doWork(Integer task) {
    int i = task.intValue();
    System.err.println(":s:: Computing task "+i);
    return new Integer(++i);
  }

}
```

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 6.26: SimpleComponentInterface.java

```java
package simpleComponent;

import org.osoa.sca.annotations.Remotable;

@Remotable
public interface SimpleComponentInterface {
  public Integer doWork(Integer task);
}
```

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 6.27: SimpleComponent.composite

```xml
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 6.28: Main.java

```java
package simpleComponent;

import org.apache.tuscany.sca.host.embedded.SCADomain;

public class Main {

  public static void main(String[] args) {
    System.err.println(":: starting ... ");
    SCADomain scaDomain = SCADomain.newInstance("SimpleComponent.
        composite");
    System.err.println(":: domain activated");
    SimpleComponentInterface simpleComp = scaDomain.getService(
        SimpleComponentInterface.class, "SimpleServiceComponent");
    System.err.println(":: got simpleComp");
    Integer x = simpleComp.doWork(new Integer(5));
    System.err.println(":: terminating ... ");
  }

}
```

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.SingleComponer` and `singleComponent.Main` class files. At line 11, we get the reference to the `SimpleComponentInterface`[3] 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 6.29: 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
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.

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

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 6.30: 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 6.31: SecondComponentInterface.java

```
package compositeComponent;

import org.osoa.sca.annotations.Remotable;

@Remotable
public interface SecondComponentInterface {
  public Integer compute(Integer task);
}
```

The `SecondComponent` *uses* a `SimpleComponent`. Therefore, in the `SecondComponent.java` implementation file, at line 7, we define a reference to a `SimpleComponentInterface`[4]

[4]as before, be careful to use the interface here, not the implementation class, otherwise

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 6.32: 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   <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.

you'll get errors

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

Bibliography

[1] Fractal ADL tutorial, 2004. http://fractal.ow2.org/tutorials/adl/index.html.

[2] ProActive manual online, 2008. http://proactive.inria.fr/release-doc/pa/multiple_html/index.html.