```
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>)`

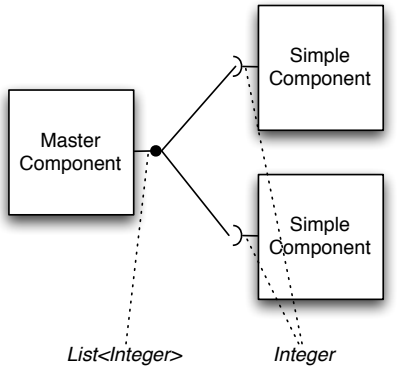

Figure 6.2: Sample configuration with two components connected to a master through a multicast port

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.

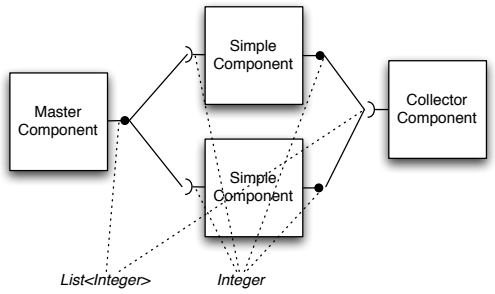

Figure 6.3: Sample configuration with two components connected to a master and a collector through a multicast and a gathercast port

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

```
<?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="completeOneWay.adl.SimpleComponent">

  <interface
    name="simple-component-interface"
    signature="completeOneWay.SimpleComponentInterface"
    role="server" />

  <interface
    name="collector-component-interface"
    signature="completeOneWay.CollectorGathercastInterface"
    role="client"
  cardinality="singleton"/>

  <content
    class="completeOneWay.SimpleComponent" />

  <controller
    desc="primitive" />

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

The test program at this point could be:

Listing 6.44: Test.java

```
package completeOneWay;

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

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

import adles2.SecondComponentInterface;

public class Test {

  /**
   * @param args
   */
  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();
```

```
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

happening 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

```
<?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="completeOneWayTerminating.adl.Composite">

  <interface
    name="outer-component-interface"
    signature="completeOneWayTerminating.MasterComponentInterface"
    role="server" />

  <interface
    name="outer-terminator-interface"
    signature="completeOneWayTerminating.TerminatorInterface"
    role="server" />

  <component
    name="master-component"
    definition="completeOneWayTerminating.adl.MasterComponent" />

  <component
    name="simple-component-1"
    definition="completeOneWayTerminating.adl.SimpleComponent" />

  <component
    name="simple-component-2"
    definition="completeOneWayTerminating.adl.SimpleComponent" />

  <component
    name="collector-component"
    definition="completeOneWayTerminating.adl.Collector" />

  <binding
    client="master-component.simple-component-interface"
    server="simple-component-1.simple-component-interface" />
  <binding
    client="master-component.simple-component-interface"
    server="simple-component-2.simple-component-interface" />

  <binding
    client="simple-component-1.collector-component-interface"
    server="collector-component.collector-component-interface" />
  <binding
    client="simple-component-2.collector-component-interface"
    server="collector-component.collector-component-interface" />

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

  <binding
    client="this.outer-terminator-interface"
    server="collector-component.collector-terminator-interface" />

```

```
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).

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.

[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

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: