ProActive v3.0.1 Documentation

ProActive v3.0.1 Documentation

Table of Contents

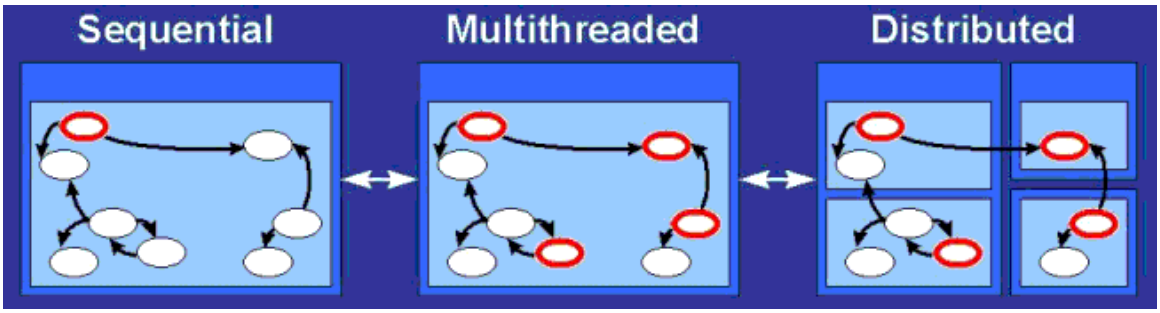
Part I. Introduction

Table of Contents

Chapter 1. Principles

1.1. Seamless sequential, multithreaded and distributed

Most of the time, activities and distribution are not known at the beginning, and change over time. **Seamless implies reuse, smooth and incremental transitions**.



A huge gap yet exists between multithreaded and distributed Java applications which forbids code reuse in order to build distributed applications from multithreaded applications. Both JavaRMI and JavaIDL, as examples of distributed object libraries in Java, put a heavy burden on the programmer because they require deep modifications of existing code in order to turn local objects into remote accessible ones. In these systems, remote objects need to be accessed through some specific interfaces. As a consequence, these distributed objects libraries do not allow polymorphism between local and remote objects. This feature is our first requirement for a metacomputing framework. It is strongly required in order to let the programmer concentrate first on modeling and algorithmic issues rather than lower-level tasks such as object distribution, mapping and load balancing.

1.2. Active objects: Unifying threads and remote objects

Given a standard object, we provide the ability to give it:

- location transparency
- activity transparency
- synchronization

1.2.1. ==> only with modifications of the instanciation code !

Three ways to transform a standard object into an active one:

1.2.2. Class-based

```
Object[] params = new Object[] { new Integer (26), "astring" };
A a = (A)
ProActive.newActive("example.A", params, node)
;
```

1.2.3. Instantiation-based

```
public class AA extends A implements Active {}
Object[] params = new Object[] { new Integer (26), "astring" };
A a = (A)
ProActive.newActive("example.AA", params, node)
;
```

1.2.4. Object-based

Allows to turn active and remote objects for which you do not have the source code; a necessary feature in the context of code mobility.

```
A a = new A (26, "astring");
a = (A)
ProActive.turnActive(a, node)
;
```

Notes:*Node allows to control the mapping*

1.3. Model of Computation: Based on previous works and studies

- **Heterogeneous model** both passive and active objects
- **Systematic asynchronous communications towards active objects**
- **No shared passive object** , Call-by-value between active objects

- **Automatic continuations** , a transparent delegation mechanism
- **wait-by-necessity** , automatic and transparent futures
- **Centralized and explicit control** , libraries of Abstractions

1.4. Reuse and Seamless: why and how do we achieve better reuse ?

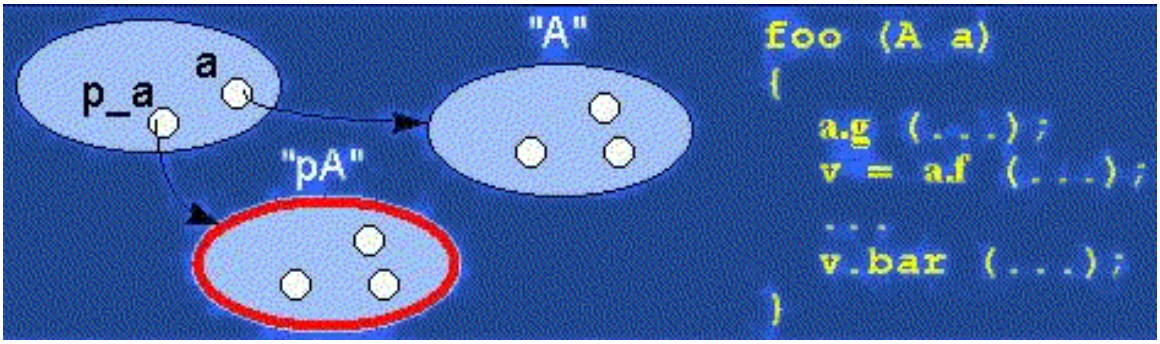
Two key features :

- **Wait-by-necessity : inter-objects synchronization**

Systematic, implicit and transparent futures. Ease the programming of synchronization and reuse of existing methods

- **Polymorphism between standard and active objects**

- Type compatibility for classes and not just for interface
- Needed and done for the future objects as well
- Dynamic mechanism (dynamically achieve if needed)



Chapter 2. ProActive Installation

ProActive is made available for download [http://www-sop.inria.fr/oasis/proactive/disclaimer.xml] under a LGPL license [http://www.gnu.org/copyleft/lesser.txt]. ProActive requires the JDK 1.4 [http://java.sun.com/j2se/1.4/] or later to be installed on your computer. Please note that ProActive will NOT run with any version prior to 1.4 since some features introduced in JDK 1.4 are essential.

2.1. Quick Start

2.1.1. To Test ProActive with the examples

- Download and unzip the ProActive archive
- Set the `JAVA_HOME` variable to the Java distribution you want to use
- Launch the scripts located in `ProActive/scripts/unix` or `ProActive/scripts/windows`
- no other setting is necessary since the scripts given with the example take care of everything

2.1.2. To develop with ProActive

- Download and unzip the ProActive archive
- Include in your `CLASSPATH` the following jar files `ProActive/ProActive.jar`, `ProActive/lib/asm.jar`, `ProActive/lib/log4j.jar`, `ProActive/lib/xercesImpl.jar`, `ProActive/lib/components/fractal.jar`, `ProActive/lib/bouncycastle.jar`
- Don't forget to launch the JVM with a security policy file [http://java.sun.com/j2se/1.3/docs/guide/security/permissions.xml]. You can also specify a log4j configuration file [http://logging.apache.org/log4j/docs/manual.xml] with the property **-Dlog4j.configuration=file:pathToFile**. If not specified a default logger that logs on the console will be created

Below are described the different steps in more details.

Part II. Programming

Table of Contents

Chapter 3. ProActive Basis, Active Object definition

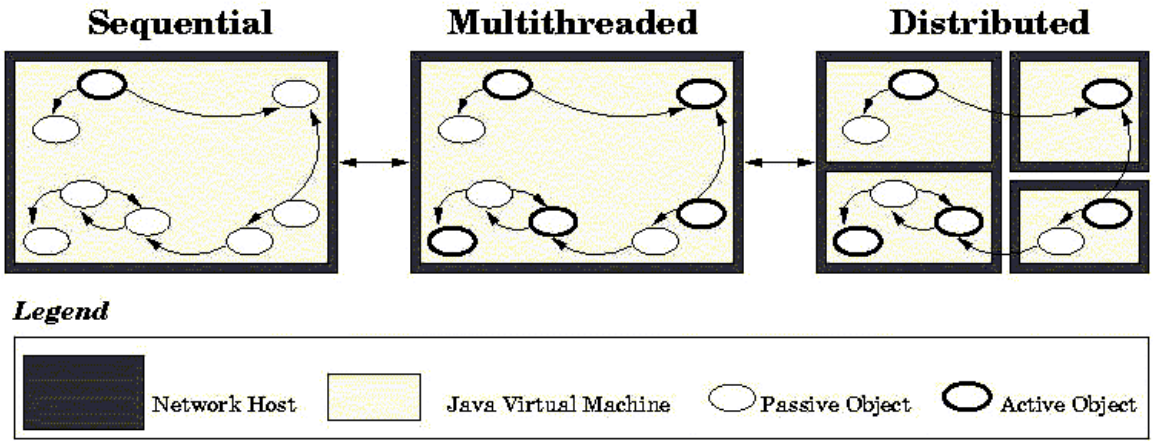
3.1. Active objects basis

Active objects are the basic units of activity and distribution used for building concurrent applications using ProActive. An active object owns its own thread. This thread only executes the methods invoked on this active object by other active objects and those of the passive objects of the subsystem that belongs to this active object. With ProActive, the programmer does not have to explicitly manipulate Thread objects, unlike in standard Java.

Active objects can be created on any of the hosts involved in the computation. Once an active object is created, its activity (the fact that it owns its own thread) and its location (local or remote) are perfectly transparent. As a matter of fact, any active object can be manipulated just like if it were a passive instance of the same class.

ProActive is a library designed for developing applications in a model introduced by the Eiffel// language. Its main features are :

- The application is structured in subsystems. There is one active object (and therefore one thread) for each subsystem and one subsystem for each active object (or thread). Each subsystem is thus composed of one active object and any number of passive objects (possibly zero). The thread of one subsystem only executes methods in the objects of this subsystem.
- There are no shared passive objects between subsystems.



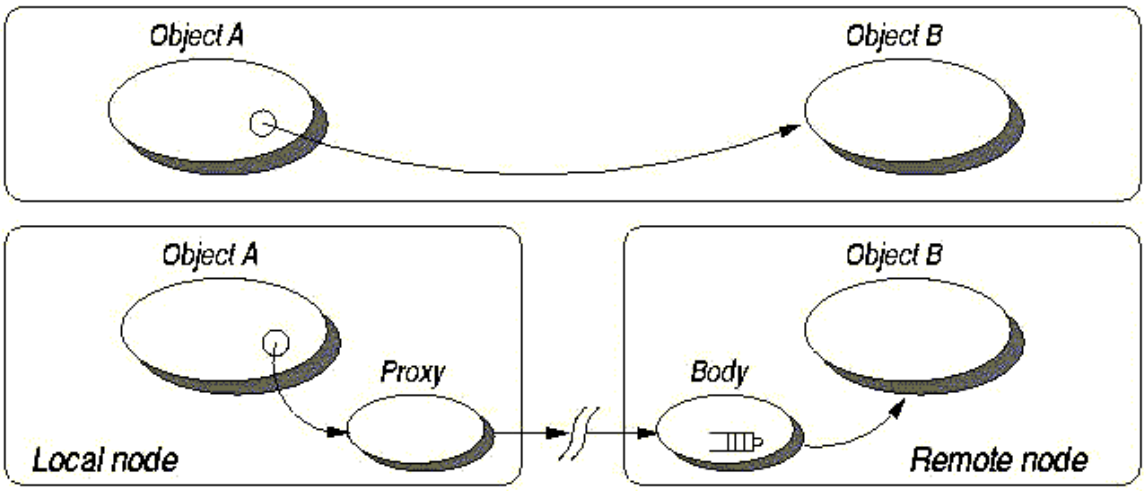
The Model

These two main features have a lot of important consequences on the topology of the application:

- Of all the objects that make up a subsystem (the active object and the passive objects), only the active object is known to objects outside of the subsystem.
- All objects (both active and passive) may have references onto active objects.
- If an object o1 has a reference onto a passive object o2, then o1 and o2 are part of the same subsystem.

This has also consequences on the semantics of message-passing between subsystems.

- When an object in a subsystem calls a method on an active object, the parameters of the call may be references on passive objects of the subsystem, which would lead to shared passive objects. This is why passive objects passed as parameters of calls on active objects are always passed by *deep-copy*. Active objects, on the other hand, are always passed by reference. Symmetrically, this also applies to objects returned from methods called on active objects.
- When a method is called on an active object, it returns immediately (as the thread cannot execute methods in the other subsystem). A *future object*, which is a placeholder for the result of the methods invocation, is returned. From the point of view of the caller subsystem, no difference can be made between the future object and the object that would have been returned if the same call had been issued onto a passive object. Then, the calling thread can continue executing its code just like if the call had been effectively performed. The role of the future object is to block this thread if it invokes a method on the future object and the result has not yet been set (i.e. the thread of the subsystem on which the call was received has not yet performed the call and placed the result into the future object): this inter-object synchronization policy is known as *wait-by-necessity*.



A call onto an active object as opposed to a call onto passive one

3.2. What is an active object

The active object is actually the composition of two objects: a *body* and a standard Java object. The body is not visible from the outside of the active object, then everything looks like if the standard object was active.

The body is responsible for receiving calls on the active object, storing these calls in a queue of pending calls (we also call *requests*. It also executes these calls in an order specified by a specific synchronization policy. If no specific synchronization policy is provided, calls are managed in a FIFO manner (first come, first served)).

Then, the thread of an active object alternatively chooses a method in the queue of pending requests and executes it. It is important to note that no parallelism is provided inside an active object. This is an important decision in the design of ProActive which enables the use of pre-post conditions and class invariants.

On the side of the subsystem which sends a call to an active object, this active object is represented by a *proxy*, whose main responsibility is to generate future objects for representing future values, transform calls into Request objects (in terms of metaobject, this is a reification) and perform deep-copy of passive objects passed as parameters.

Chapter 4. Typed Group Communication

4.1. Overview

Group communication is a crucial feature for high-performance and Grid computing. While previous works and libraries proposed such a characteristic (e.g. MPI, or object-oriented frameworks), the use of groups imposed specific constraints on programmers, for instance the use of dedicated interfaces to trigger group communications.

We aim at a more flexible mechanism. We propose a scheme where, given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remains typed.

In order to ease the use of the group communication, we provide a set of static methods on the `ProActiveGroup` class and a set of methods on the `Group` interface.

Here, a short compilation about the syntax and some method used in the Group Communication API is presented. More informations follow.

```
  // created at once,
  // with parameters specified  in params,
  // and on the nodes specified in nodes
A ag1 = (A) ProActiveGroup.newGroup( "A", params, [nodes]);
  // A general group communication without result
  // A request to foo is sent in parallel to all active objects
  // in the target group (ag1)
ag1.foo(...);
  // A general group communication with a result
V vg = ag1.bar(...);
  // vg is a typed group of "V": operation
  // below is also a collective operation
  // triggered on results
vg.f1();
```

4.2. Creation of a Group

Any object that is reifiable has the ability to be included in a group. Groups are created using the static method `ProActiveGroup.newGroup`. The common superclass for all the group members has to be specified, thus giving the group a minimal type.

Let us take a standard Java class:

```
class A {
  public A() {}
  public void foo (...) {...}
  public V bar (...) {...}
  ...
}
```

Here are examples of some group creation operations:

```
  // Pre-construction of some parameters:
  //  For constructors:
Object[][] params = {{...} , {...} , ... };
  //  Nodes to identify JVMs to map objects
Node[] nodes  = { ... , ..., ... };
  // Solution 1:
  // create an empty group of type "A"
A ag1 = (A) ProActiveGroup.newGroup("A");
  // Solution 2:
  // a group of type "A" and its members are
  // created at once,
  // with parameters specified  in params,
  // and on the nodes specified in nodes
A ag2 = (A) ProActiveGroup.newGroup("A", params, nodes);
  // Solution 3:
  // a group of type "A"  and its members are
  // created at once,
  // with parameters specified in params,
  // and on the nodes directly specified
A ag3 = (A) ProActiveGroup.newGroup("A", params[],
              {rmi://globus1.inria.fr/Node1,
               rmi://globus2.inria.fr/Node2});
```

Elements can be included into a typed group only if their class equals or extends the class specified at the group creation. For example, an object of class B (B extending A) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

4.3. Group representation and manipulation

The **typed group representation** we have presented corresponds to the functional view of groups of objects. In order to provide a dynamic management of groups, a second and complementary representation of a group has been designed. In order to manage a group, this second representation must be used instead. This second representation, **the management representation**, follows a more standard pattern for grouping objects : the `Group` interface.

We are careful to have a strong coherence between both representations of the same group, which implies that modifications executed through one representation are immediately reported on the other one. In order to switch from one representation to the other, two methods have been defined : the static method named `ProActiveGroup.getGroup`, returns the Group form associated to the given group object; the method `getGroupBytype` defined in the `Group` interface does the opposite.

Below is an example of when and how to use each representation of a group:

```
  // definition of one standard Java object
  // and two active objects
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b  = (B) ProActive.newActive("B", paramsB[], node);
  // Note that B extends A
```

```
  // For management purposes, get the representation
  // as a group given a typed group, created with
  // code on the left column:
Group gA = ProActiveGroup.getGroup(ag1);
  // Now, add objects to the group:
  // Note that active and non-active objects
  // may be mixed in groups
gA.add(a1);
gA.add(a2);
gA.add(b);
  // The addition of members to a group immediately
  // reflects on the typed group form,  so a method
  // can be invoked on the typed group  and will
  // reach all its current members
ag1.foo(); // the caller of ag1.foo() may not belong to ag1
  // A new reference to the typed group
  // can also be built as follows
A ag1new = (A) gA.getGroupByType();
```

4.4. Group as result of group communications

The particularity of our group communication mechanism is that the **result** of a typed group communication **is also a group**. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results. Nevertheless, the result group can be immediately used to execute another method call, even if all the results are not available. In that case the **wait-by-necessity** mechanism implemented by ProActive is used.

```
  // A method call on a group, returning a result
V vg = ag1.bar();
  // vg is a typed group of "V": operation
  // below is also a collective operation
  // triggered on results
vg.f1();
```

As said in the Group creation section, groups whose type is based on final classes or primitive types cannot be built. So, the construction of a dynamic group as a result of a group method call is also limited. Consequently, only methods whose return type is either void or is a 'reifiable type', in the sense of the Meta Object Protocol of ProActive, may be called on a group of objects; otherwise, they will raise an exception at run-time, because the transparent construction of a group of futures of non-reifiable types fails.

To take advantage with the asynchronous remote method call model of ProActive, some new synchronization mechanisms have been added. Static methods defined in the `ProActiveGroup` class enable to execute various forms of synchronisation. For instance: `waitOne`, `waitN`, `waitAll`, `waitTheNth`, `waitAndGet`. Here is an exemple :

```
  // A method call on a typed group
V vg = ag1.bar();
  // To wait and capture the first returned
  // member of vg
```

```
V v = (V) ProActiveGroup.waitAndGetOne(vg);
  // To wait all the members of vg are arrived
ProActiveGroup.waitAll(vg);
```

4.5. Broadcast vs Dispatching

Regarding the parameters of a method call towards a group of objects, the default behaviour is to broadcast them to all members. But sometimes, only a specific portion of the parameters, usually dependent of the rank of the member in the group, may be really useful for the method execution, and so, parts of the parameter transmissions are useless. In other words, in some cases, there is a need to transmit different parameters to the various members.

A common way to achieve the scattering of a global parameter is to use the rank of each member of the group, in order to select the appropriate part that it should get in order to execute the method. There is a natural traduction of this idea inside our group communication mechanism:**the use of a group of objects in order to represent a parameter of a group method call that must be scattered to its members**.

The default behaviour regarding parameters passing for method call on a group, is to pass a deep copy of the group of type P to all members. Thus, in order to scatter this group of elements of type P instead, the programmer must apply the static method `setScatterGroup` of the `ProActiveGroup` class to the group. In order to switch back to the default behaviour, the static method `unsetScatterGroup` is available.

```
  // Broadcast the group gb to all the members
  // of the group ag1:
ag1.foo(gb);
  // Change the distribution mode of the
  // parameter group:
ProActiveGroup.setScatterGroup(gb);
  // Scatter the members of gb onto the
  // members of ag1:
ag1.foo(gb);
```

To learn more, see the JavaDoc [../../../../index.xml] and the paper **Efficient, Flexible and Typed Group Communications for Java** located in the Documentation page [http://www-sop.inria.fr/oasis/ProActive/doc/index.xml].

Chapter 5. OOSPMD

5.1. OOSPMD : Introduction

SPMD stands for Single Program Multiple Data. Merged into an object-oriented framework, an SPMD programming model becomes an OOSPMD programming model.

The typed group communication system can be used to simulate MPI-style collective communication. Contrary to MPI that requires all members of a group to collectively call the same communication primitive, our group communication scheme let the possibility to one activity to call a method on the group.

The purpose of the our group communication is to free the programmer from having to implement the complex communication code required for setting identical group in each SPMD activity, group communication, thereby allowing the focus to be on the application itself.

This page presents the mechanism of typed group communication as an new alternative to the old SPMD programming model. While being placed in an object-oriented context, this mechanism helps the definition and the coordination of distributed activities. The approach offers, through modest size API, a better structuring flexibility and implementation. The automation of key communication mechanisms and synchronization simplifies the writing of the code.

The main principle is rather simple: an spmd group is a group of active objects where each one has a group referencing all the active objects.

5.2. SPMD Groups

An spmd group is a ProActive typed group built with the **ProSPMD.newSPMDGroup()** method. This method looks like the **ProActiveGroup.newGroup()**; they have similar behavior (and overloads). The difference is that each members of an spmd group have a reference to a group containing all the others members and itself (i.e. a reference to the spmd group itself).

Given a standard Java class:

```
class A {
  public A() {}
  public void foo (...) {...}
  public void bar (...) {...}
  ...
}
```

The spmd group is built as follow:

```
Object[][] params = {{...} , {...} , ... };
Node[] nodes  = { ... , ..., ... };
A agroup = (A) ProSPMD.newSPMDGroup("A", params[], nodes);
```

Object members of an spmd group are aware about the whole group. They can obtain some informations about the spmd group they belong to such as the size of the group, their rank in the group, and a reference to the group in order to get more informations or to communicate with method invocations. Those informations are respectively obtained using the static methods **getMySPMDGroupSize()**, **getMyRank()**, and **getSPMDGroup()** of the **ProSPMD** class.

5.3. Barrier : Introduction

ProActive provides three kinds of barrier to synchronize activities. The feature is specially useful in a SPMD programming style. A barrier stops the activity of the active object that invokes it until a special condition is satisfied. Notice that, as the opposite of MPI or such libraries, the ProActive barriers do not stop the current activity immediately (when the **barrier** method is encountered). The current method actually keeps on executing until the end. The barrier will be activated at the end of the service: no other service will be started until all the AOs involved in the barrier are at that same point.

The three barriers are named:

- the **Total Barrier**
- the **Neighbor Barrier**
- the **Method-based Barrier**

Here is a presentation about how to use those barriers.

5.4. Total Barrier

Total barrier directly involves the spmd group. A call to **barrier(String)** will block until all the members in the spmd group have themselves reach and called the identical **ProSPMD.barrier()** primitive. A call communicates with all the members of the spmd group. The barrier is released when the Active Object has received a *barrier message* from all other members of the spmd group (including itself).

The string parameter is used as unique identity name for the barrier. It is the programmer responsibility to ensure that two (or more) different barriers with the same id name are not invoked simultaneously.

Let us take a Java class that contains a method calling a total barrier, here the method **foo**:

```
class A {
  public A() {}
  public void foo (...) {
    ...
    ProSPMD.barrier("MyBarrier");
  }
  public void bar (...) {...}
  ...
}
```

Note that usually, strings used as unique ID are more complex; they can be based on the full name of the class or the package (**org.objectweb.proactive.ClassName**), for example. The spmd group is built as follow:

```
Object[][] params = {{...} , {...} , ... };
Node[] nodes  = { ... , ..., ... };
A agroup = (A) ProSPMD.newSPMDGroup("A", params[], nodes);
```

Here the main method of our application:

```
agroup.foo();
agroup.bar();
```

The call to **barrier** launched by all members (in the invocation of **foo**) ensures that no one will initiate the **bar** method before all the **foo** methods end.

The programmer have to ensure that **all the members of an spmd group call the barrier method** otherwise the members of the group may indefinitely wait.

5.5. Neighbor barrier

The Neighbor barrier is a kind of light weighted barrier, involving not all the member of an spmd group, but only the Active Objects specified in a given group.

barrier(String,group) initiates a barrier only with the objects of the specified group. Those objects, that contribute to the end of the barrier state, are called *neighbors* as they are usually local to a given topology, An object that invoke the Neighbor barrier HAVE TO BE IN THE GROUP given as parameter. The *barrier message* is only sent to the group of neighbors.

The programmer has to explicitly build this group of neighbors. The topology API can help him or her to build such group. Topologies are groups. They just give special access to their members or (sub)groups members. For instance, a matrix fits well with the topology **Plan** that provides methods to get the reference of neighbor members (**left**, **right**, **up**, **down**). See the javadoc of the topology package [../../../../org/objectweb/proactive/core/group/topology/package-summary.xml] for more informations.

Like for the Total barrier, the string parameter represents a unique identity name for the barrier. The second parameter is the group of neighbors built by the programmer. Here is an example:

```
ProSPMD.barrier("MyString", neighborGroup);
```

Refer to the *Jacobi* example to see a use of the Neighbor barrier. Each submatrix needs only to be synchronized with the submatrixes which it is in contact.

This barrier increases the asynchronism and reduce the amount of exchanged messages.

5.6. Method Barrier

The Method barrier does no more involve extra messages to communicate (i.e. the *barrier messages*). Communications between objects to release a barrier are achieved by the standard method call and request reception of ProActive.

The method **barrier(String[])** stops the active object that calls it, and wait for a request on the specified methods to resume. The array of string contains the name of the awaited methods. The order of the methods does not matter. For example :

```
ProSPMD.barrier({"foo", "bar", "gee"});
```

The caller will stop and wait for the three methods. bar or gee can came first, then foo. If one wants wait for foo, then wait for bar, then wait for gee, three calls can be successively done:

```
ProSPMD.barrier({"foo"});
ProSPMD.barrier({"bar"});
ProSPMD.barrier({"gee"});
```

A method barrier is used without any group (spmd or not). To learn more, browse the JavaDoc [../../../../index.xml].

Chapter 6. Active Object Migration

6.1. Migration Primitive

The migration of an active object can be triggered by the active object itself, or by an external agent. In both cases a single primitive will eventually get called to perform the migration. It is the method `migrateTo` accessible from a migratable body (a body that inherits from `AbstractMigratableBody`).

In order to ease the use of the migration, we provide 2 sets of static methods on the `ProActive` class. The first set is aimed at the migration triggered from the active object that wants to migrate. The methods rely on the fact that the calling thread **is** the active thread of the active object :

- `migrateTo(Object o)` : migrate to the same location as an existing active object
- `migrateTo(String nodeURL)` : migrate to the location given by the URL of the node
- `migrateTo(Node node)` : migrate to the location of the given node

The second set is aimed at the migration triggered from another agent than the target active object. In this case the external agent must have a reference to the Body of the active object it wants to migrate.

- `migrateTo(Body body, Object o, boolean priority)` : migrate to the same location as an existing active object
- `migrateTo(Body body, String nodeURL, boolean priority)` : migrate to the location given by the URL of the node
- `migrateTo(Body body, Node node, boolean priority)` : migrate to the location of the given node

6.2. Using migration

Any active object has the ability to migrate. If it references some passive objects, they will also migrate to the new location. Since we rely on the serialization to send the object on the network, **the active object must implement the serializable interface**. To migrate, an active object must have a method which contains a call to the migration primitive. This call must be the last one in the method, i.e the method must return immediately after. Here is an example of a method in an active object :

```
public void moveTo(String t) {
  try {
    ProActive.migrateTo(t);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

We don't provide any test to check if the call to migrateTo is the last one in the method, hence if this rule is not enforced, it can lead to unexpected behavior. Now to make this object move, you just have to call its `moveTo()` method.

6.3. Complete example

```
 import org.objectweb.proactive.ProActive;
 public class SimpleAgent implements Serializable {
  public SimpleAgent() {
  }
  public void moveTo(String t) {
   try {
    ProActive.migrateTo(t);
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
  public String whereAreYou() {
   try {
    return InetAddress.getLocalHost().getHostName();
   } catch (Exception e) {
    return "Localhost lookup failed";
   }
  }
  public static void main (String[] args) {
   if (!(args.length>0)) {
    System.out.println("Usage: java
migration.test.TestSimple hostname/N\
odeName ");
    System.exit(-1);
   }
   SimpleAgent t = null;
   try {
    // create the SimpleAgent in this JVM
    t = (SimpleAgent)
ProActive.newActive("migration.test.SimpleAgent",nu\
ll);
   } catch (Exception e) {
    e.printStackTrace();
   }
   // migrate the SimpleAgent to the location identified by the
given no\
de URL
   // we assume here that the node does already exist
   t.moveTo(args[0]);
   System.out.println("The Active Object is now on host " +
t.whereAreYou(\
));
  }
 }
```

The class SimpleAgent implements `Serializable` so the objects created will be able to migrate. We need to provide an empty constructor to avoid side effects during the creation of active objects. This object has two methods, `moveTo()` which makes it migrate to the specified location, and `whereAreYou()` which returns the hostname of the new location of the agent.

In the main method, we first need to create an active object, which is done through the call to `newActive()`. Once this is done, we can call methods on it as on any object. We call its `moveTo` method which will make it migrate to the node specified as parameter and then we ask it what is its current location.

6.4. Dealing with non-serializable attributes

The migration of an active object uses the serialization. Unfortunately, not all the objects in the Java language are serializable. We are going to see a simple method to deal with such attributes in the case their value does not need to be saved. For more complex cases, the reader can have a look to the Java RMI specifications.

When a `NotSerializable` exception is thrown, the first step to solve the problem is to identify the variable responsible, i.e the one which is not serializable... In front of the declaration of this variable, put the keyword `transient`. This indicates that the value of this variable should not be serialized. After the first migration, this field will be set to null since it has not been saved. So we have to rebuild it upon arrival of the active object on its new location. This can easily be done by providing in the active object the standard method

```
   private void readObject(java.io.ObjectInputStream in) throws
java.io.IO\
Exception, ClassNotFoundException;
```

See the `Serializable` interface in the standard JavaDoc to learn more.

Chapter 7. Exception Handling

7.1. Exceptions and Asynchrony

In the asynchronous environment provided by ProActive, exceptions cannot be handled the same as in a sequential environment. Let's see the problem with exceptions and asynchrony in a piece of code:

```
1
try {
2    Result r = someAO.someMethodCall(); // Asynchronous method call that \
can throw an exception
3    // ...
4    doSomethingWith(r);
5 }
catch (SomeException se) {
6    doSomethingWithMyException(se);
7 }
```

In this piece of code, as the method call in line 2 is asynchronous, we don't wait for its completion and continue the execution. So, it's possible the control flow exits the **try**. In this case, if the method call ends up with an exception, we cannot throw it anymore back in the code because we are no more in the **try** block. That's why, by default, ProActive method calls with potential exceptions are handled synchronously.

7.1.1. Barriers around try blocks

The ProActive solution to this problem is to put barriers around **try**/**catch** blocks. This way, the control flow cannot exit the block, the exception can be handled in the appropriate **catch** block, and the call is asynchronous within the block.

With this configuration, the potential exception can be throw for several points:

- When accessing a future
- In the barrier
- Using the provided API (see after)

Let's reuse the previous example to see how to use these barriers

```
1
ProActive.tryWithCatch(SomeException.class);
2
try {
3     Result r = someAO.someMethodCall(); // Asynchronous method call tha\
t can throw an exception
4     // ...
5     doSomethingWith(r);
6
ProActive.endTryWithCatch();
7   }
catch (SomeException se) {
8     doSomethingWithMyException(se);
9   }
```

```
finally {
10
ProActive.removeTryWithCatch();
11
}
```

With this code, the call in line 3 will be asynchronous, and the exception will be handled in the correct **catch** block. Even if this implies waiting at the end of the **try** block for the completion of the call.

Let's see in detail the needed modifications to the code:

- ProActive.tryWithCatch() call right before the **try** block. The parameter is either the caught exception class or an array of these classes if there are many
- ProActive.endWithTry() at the end of the **try** block
- ProActive.removeTryWithCatch() at the beginning of the **finally** block, so the block becomes mandatory

7.1.2. TryWithCatch Annotator

These needed annotations can be seen as cumbersome, so we provide a tool to add them automatically to a given source file. It transforms the first example code in the second. Here is a sample session with the tool:

```
$ ProActive/scripts/unix/trywithcatch.sh MyClass.java # A backup will be ma\
de in MyClass.java~
--- ProActive TryWithCatch annotator -----------------------
$ diff -u MyClass.java~ MyClass.java
--- MyClass.java~
+++ MyClass.java
@@ -1,9 +1,13 @@
 public class MyClass {
    public MyClass someMethod(AnotherClass a) {
+       ProActive.tryWithCatch(AnException.class);
        try {
           return a.aMethod();
+          ProActive.endTryWithCatch();
        } catch (AnException ae) {
           return null;
+       } finally {
+          ProActive.removeTryWithCatch();
        }
    }
 }
```

As we can see, ProActive method calls are added to make sure all calls within **try**/**catch** blocks are handled asynchronously.

7.1.3. Additional API

We have seen the 3 methods mandatory to perform asynchronous calls with exceptions, but the complete API includes two more calls. So far, the blocks boundaries define the barries. But, some control over the barrier is provided thanks to two additional methods.

The first method is ProActive.throwArrivedException(). During a computation an exception may be raised but there is no point from where the exception can be thrown (a future or a barrier). The solution is to call the ProActive.throwArrivedException() method which simply queries ProActive to see if an exception has arrived with no opportunity of being thrown back in the user code. In this case, the exception is thrown by this method.

The method behaviour is thus dependant on the timing. That is, calling this method may or may not result in an exception being thrown, depending on the time for an exception to come back. That's why another method is provided, this is ProActive.waitForPotentialException(). Unlike the previous one, this method is blocking. After calling this method, either an exception is thrown, or it is assured that all previous calls in the block completed successfully, so no exception can be thrown from the previous calls.

7.2. Non-Functional Exceptions

7.2.1. Overview

In the first part, we were concerned with functional exception. That is, exceptions originating from "business" code. The middleware adds its set of exceptions that we call Non-Functional Exceptions (NFE): network errors, ... ProActive has a mechanism for dealing with these exceptions.

7.2.2. Exception types

We have classified the non functional exceptions in two categories: those on the proxy, and those on the body. So, exceptions concerning the proxy are in the org.objectweb.proactive.core.exceptions.proxy package and inherits from the ProxyNonFunctionalException package.

7.2.3. Exception handlers

The NFE mechanism in ProActive calls user defined handlers when a NFE is thrown. A handler implements the following interface:

```
public interface NFEListener {
   public boolean handleNFE(NonFunctionalException e);
}
```

The handleNFE method is called with the exception to handle as parameter. The boolean return code indicates if the handler could do something with the exception. This way, if no handler could do anything with a given exception, the default behavior is used.

If the exception is on the proxy side, the default behaviour is to throw the exception which is a RuntimeException. But on the proxy side, the default behaviour is to log the exception with its stack trace to avoid killing an active object.

7.2.3.1. Association

These handlers are associated to entities generating exceptions. These are: an active object proxy, a body, a JVM. Given a NFE, the handlers on the local JVM will be executed, then either those associated to the proxy or the body depending on the exception.

Here is an example about how to add a handler to an active object on its side (body):

```
ProActive.addNFEListenerOnAO(myAO,
new NFEListener() {
```

```
public boolean handleNFE(NonFunctionalException nfe) {

// Do something with the exception...

// Return true if we were able to handle it

return true;
   }
});
```

Handlers can also be added to the client side (proxy) of an active object with

```
ProActive.addNFEListenerOnProxy(ao, handler)
```

or to a JVM with

```
ProActive.addNFEListenerOnJVM(handler)
```

and even to a group with

```
ProActive.addNFEListenerOnGroup(group, handler)
```

These handlers can also be removed with

```
ProActive.removeNFEListenerOnAO(ao, handler),
ProActive.removeNFEListenerOnProxy(ao, handler),
ProActive.removeNFEListenerOnJVM(handler)
ProActive.removeNFEListenerOnGroup(group, handler)
```

It’s also possible to define an handler only for some exception types, for example:

```
ProActive.addNFEListenerOnJVM(
new TypedNFEListener(
   SendRequestCommunicationException.
class,

new NFEListener() {

public boolean handleNFE(NonFunctionalException e) {

// Do something with the SendRequestCommunicationException...

// Return true if we were able to handle it

return true;
   }
```

```
}));
```

You can use NFE for example, to automatically remove dead elements from a ProActive group when trying to contact them. This can be achieved using the following construction:

```
ProActive.addNFEListenerOnGroup(group,↩
FailedGroupRendezVousException.AUTO_\
GROUP_PURGE);
```

Note that this currently works only for one-way calls.

Chapter 8. Branch and Bound API

The outline of this short handbook:

1. Overview
2. The API Architecture
3. The API Description
4. An Example: FlowShop
5. Future Work

8.1. Overview

The Branch and Bound (BnB) consists to an algorithmic technique for exploring a solution tree from which returns the optimal solution.

The main goal of this BnB API is to provide a set of tools for helping the developpers to parallelize his BnB problem implementation.

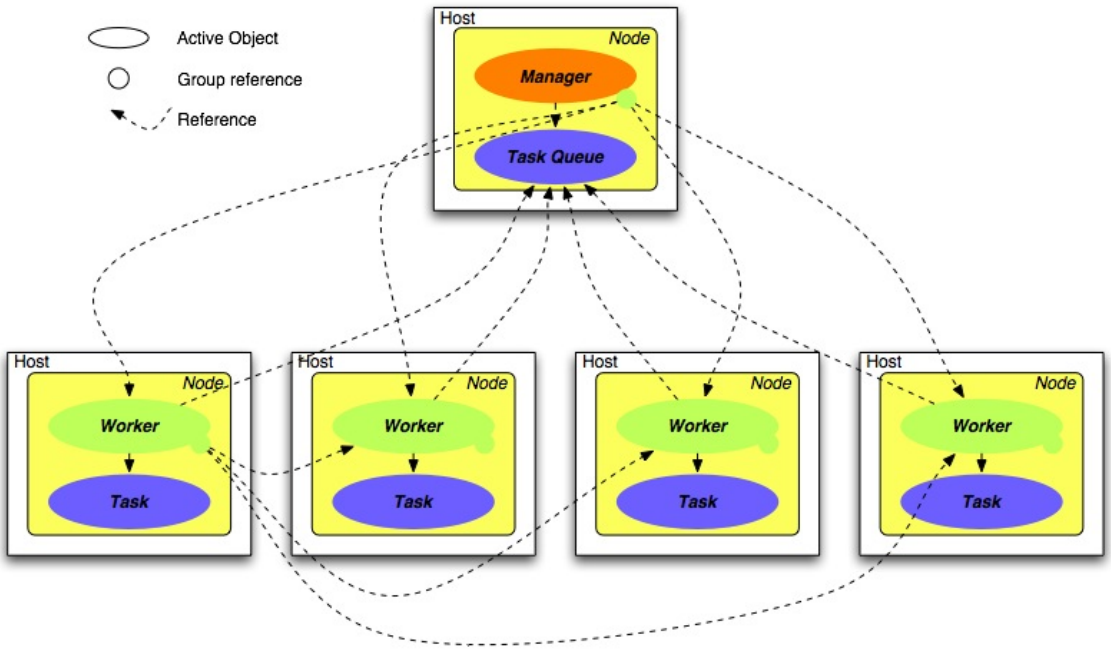
The main features are:

- Hidding computation distribution.
- Dynamic task splitting.
- Automatic solution gathering.
- Task communications for broadcasting best current solution.
- Different behaviors for task allocation, provided by the API or yourself.
- Open API for extensions.

Further research information is available here [http://www-sop.inria.fr/oasis/personnel/Alexandre.Di_Costanzo/publications.xml].

8.2. The Model Architecture

The next figure show the architecture of the API:

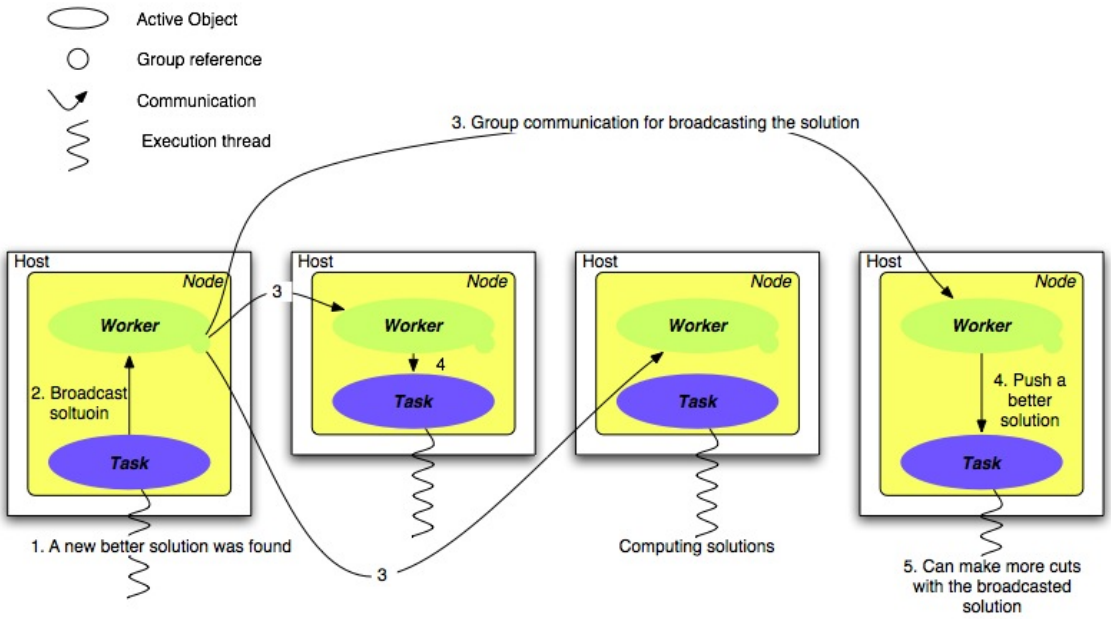


The API architecture.

The API active objects are:

- **Manager** : the main point of the API. It is the master for deploying and managing Workers. Also, it attributes Tasks to free workers. The Tasks are provided the Task Queue.
- **Task Queue** : provides Task in a specific order to the Manager.
- **Worker** : broadcasts solution to all Task, and provides the API environment to the Tasks.
- **Task** : the user code to compute.

All Workers have a group reference on all the others. The next figure show step by step how a Task can share a new better solution with all:



Broadcasting a new solution.

Finally, the methods order execution:

1. rootTask.initLowerBound(); // compute a first lower bound
2. rootTask.initUpperBound(); // compute a first upper bound
3. Vector splitted = rootTask.split(); // generate a set of tasks
4. for i in splitted do in parallel

   splitted[i].initLowerBound();

   splitted[i].initUpperBound();

   Result ri = splitted.execute()
5. Result final = rootTask.gather(Result[] ri); // gathering all result

Keep in mind that is only "initLower/UpperBound" and "split" methods are called on the root task. The "execute" method is called on the root task's splitted task.

8.3. The API Details

8.3.1. The Task Description

The **Task** object is located in this followed package:

```
org.objectweb.proactive.branchnbound.core
```

All abstract methods are described bellow:

8.3.1.1. public Result execute()

It is the place where the user has to put his code for solving a part and/or the totality of his BnB problem. There are 2 main usages of it. The first one consists to divide the task and returning no result. The second is to try to improve the best solution.

8.3.1.2. public Vector split()

This is for helping the user when he wants to divide a task. In a future work we have planned to use this method in an automatic way.

8.3.1.3. public void initLowerBound()

Initialize a lower bound local to the task.

8.3.1.4. public void initUpperBound()

Initialize a upper bound local to the task.

8.3.1.5. public Result gather(Result[] results)

This one is **not abstract** but it is strongly recommended to override it. The default behavior is to return the smallest Result gave by the compareTo method. That's why it is also recommended to override the **compareTo(Object)** method.

Some class variables are provided by the API to help the user for keeping a code clear. See next their descriptions:

```
protected Result initLowerBound; // to store the lower bound
protected Result initUpperBound; // to store the upper bound
protected Object bestKnownSolution; // setted automaticaly by the API
                        // with the best current solution
protected Worker worker; // to interact with the API (see after)
```

From the Task, specialy within the execute() method, the user has to interact with the API for sending sub-tasks, which result from a split call, to the task queue, or broadcasting to other tasks a new better solution, etc.

The way to do that is to use the class variable: **worker**.

- Broadcasting a new better solution to all the other class:

```
this.worker.setBestCurrentResult(newBetterSolution);
```

- Sending a set of sub-tasks for computing:

```
this.worker.sendSubTasksToTheManager(subTaskList);
```

- For a smarter split, checking that the task queue needs more tasks:

```
BooleanWrapper workersAvailable = this.worker.isHungry();
```

8.3.2. The Task Queue Description

This manages the task allocation. The main functions are: providing tasks in a sepcial order, and keeping results back.

For the moment, there are 2 different queue types provided by the API:

- **BasicQueueImpl** : provides tasks in FIFO order.
- **LargerQueueImpl** : provides tasks in a larger order, as Breadth First Search algorithm.

By extending the **TaskQueue** you can use a specialized task allocator for your need.

8.3.3. The ProActiveBranchNBound Description

Finally, it is the main entry point for starting, and controlling your computation.

```
Task task = new YourTask(someArguments);
Manager manager =  ProActiveBranchNBound.newBnB(task,
                  nodes,
                  LargerQueueImpl.class.getName());
Result futureResult = manager.start(); // this call is asynchronous
```

Tip: use the constructor **ProActiveBranchNBound.newBnB(Task, VirtualNode[], String)** and **do not activate** virtual nodes. This method provides a faster deployment and active objects creation way. Communications between workers are also optimized by a hierarchic group based on the array of virtual nodes. That means when it is possible define a virtual node by clusters.

8.4. An Example: FlowShop

This example solves the permutation flowshop scheduling problem, with the monoobjective case. The main objective is to minimized the overall completion time for all the jobs, i.e. makespan. A flowshop problem can be represented as a set of n jobs; this jobs have to scheduled on a set of m machines. Each jobs is defined by a set of m distinct operations. The goal consists to determine the sequence used for all machines to execute operations.

The algorithm used to find the best solution, tests all permutations and try to cut bad branches.

Firstly, the **Flowshop Task**:

```
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.branchnbound.core.Result;
import org.objectweb.proactive.branchnbound.core.Task;
import
↪
org.objectweb.proactive.branchnbound.core.exception.NoResultsExcepti\
on;
public class FlowShopTask extends Task {
 public FlowShopTask() {
  // the empty no args constructor for ProActive
 }
 /**
  * Contruct a Task which search solution for all permutations to the
  * Flowshop problem. Use it to create the root Task.
  */
 public FlowShopTask(FlowShop fs) {
  this.flowshopProblem = fs;
 }
}
```

Now, implement all Task abstract methods.

Computation **bound** methods:

```
// Compute the lower bound
```

```
public void initLowerBound() {
  this.lowerBound = this.computeLowerBound(this.fs);
}
// Compute the upper bound
public void initUpperBound() {
  this.upperBound = this.computeUpperBound(this.fs);
}
```

The **split** method:

```
public Vector split() {
  // Divide the set of permutations in 10 sub-tasks
  int nbTasks = 10;
  Vector tasks = new Vector(nbTasks);
  for (int i = 0 ; i < nbTasks ; i++){
    tasks.add(new FlowShopTask(this, i, nbTasks));
  }

  return tasks;
}
```

Then, the **execute** method:

```
public Result execute() {

  if (! this.iHaveToSplit()) {
  // Test all permutation
    while((FlowShopTask.nextPerm(currentPerm)) != null) {
       int currentMakespan;
       fsr.makespan = ((FlowShopResult)this.bestKnownSolution).makespan;
       fsr.permutation = ((FlowShopResult)this.bestKnownSolution).permutat\
ion;
       if ((currentMakespan = FlowShopTask.computeConditionalMakespan(
          fs, currentPerm,
          ((FlowShopResult) this.bestKnownSolution).makespan,
          timeMachine)) < 0) {
       //bad branch
         int n = currentPerm.length + currentMakespan;
         FlowShopTask.jumpPerm(currentPerm, n, tmpPerm[n]);
         // ...
      } else {
        // better branch than previous best
        fsr.makespan = currentMakespan;
        System.arraycopy(currentPerm, 0, fsr.permutation, 0,
                         currentPerm.length);
        r.setSolution(fsr);
        this.worker.setBestCurrentResult(r);
      }
    }
  } else {
  // Using the Stub for an asynchronous call
    this.worker.sendSubTasksToTheManager(
     ((FlowShopTask) ProActive.getStubOnThis()).split());
```

```
  }

      // ...

  r.setSolution(bestKnownSolution);
  return r;
}
```

This example is available in a complete version here [http://www-sop.inria.fr/oasis/ProActive/apps/flowshop.xml].

8.5. Future Work

- An auto-dynamic task splitting mechanism.
- Providing more queues for task allocation.
- A new task interface for wrapping native code.

Part III. Deploying

Table of Contents

Chapter 9. ProActive Basic Configuration

9.1. Overview

In order to get easier and more flexible configuration in ProActive, we introduced an xml file where all ProActive related configuration is located. It represents properties that will be added to the System when an application using ProActive is launched. Some well-known properties(explained after) will determine the behaviour of ProActive services inside a global application. That file can also contain **user-defined** properties to be used in their application.

9.2. How does it work ?

Using this file is very straightforward, since all lines must follow the model: <prop **key**="somekey" **value**="somevalue"/>

Those properties will be set in the System using System.setProperty(key,value) **if and only if** this property is not already set in the System.

If an application is using ProActive, that file is loaded once when a method is called through a ProActive "entry point". By "entry point" we mean ProActive class, NodeFactory class, RuntimeFactory class (static block in all that classes).

For instance calling **ProActive.newActive or NodeFactory.getNode** .... loads that file. This only occurs once inside a jvm.

As said before this file can contain **user-defined** properties. It means that people used to run their application with:

java .... -Dprop1=value1 -Dprop2=value2 .... -Dpropn=valuen can define all their properties in the ProActive configuration file with:

<prop **key**="prop1" **value**="value1"/>

<prop **key**="prop2" **value**="value2"/>

...

<prop **key**="propn" **value**="valuen"/>

9.3. Where to access this file ?

There is a default file with default ProActive options located under ProActive/src/org/objectweb/proactive/core/config/ProActiveConf
This file is automatically copied with the same package structure under the classes directory when compiling source files with the ProActive/compile/build facility. Hence it is included in the jar file of the distribution under org/objectweb/proactive/core/config/ProActiveConfiguration.xml (See below for default options).

People can specify their own configuration file by running their application with proactive.configuration option, i.e

java ... **-Dproactive.configuration=pathToTheConfigFile**. In that case, the given xml file is loaded. Some ProActive properties(defined below) are required for applications using ProActive to work, so even if not defined in user config file, they will be loaded programatically with default values. So people can just ignore the config file

if they are happy with the default configuration or create their own file if they want to change ProActive properties values or add their own properties

A specific tag: **<ProActiveUserPropertiesFile>** is provided in Deployment Descriptor [Descriptor.xml] to notify remote jvms which configuration file to load once created:

```
<jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcesss">
...
 <ProActiveUserPropertiesFile>
 <absolutePath value="/net/home/rquilici/config.xml"/>
 </ProActiveUserPropertiesFile>
...
</jvmProcess>
```

9.4. ProActive properties

- **Required**

  - **proactive.communication.protocol** represents the communication protocol i.e the protocol, objects on remote JVMS are exported with. At this stage several protocols are supported: **RMI(rmi), HTTP(http), IBIS/RMI(ibis), SSH tunneling for RMI/HTTP(rmissh), JINI(jini)**. It means that once the JVM starts, Nodes, Active Objects that will be created on this JVM, will export themselves using the protocol specified in **proactive.communication.protocol** property. They will be reachable transparently through the given protocol.
  - **schema.validation** . Two values are possible:**enable, disable**. If enable, all xml files will be validated against provided schema. Default is **disable**
  - **proactive.future.ac** . Two values are possible:**enable, disable** If enable, automatic continuations [AC.xml] are activated. Default is **enable**

  Note that if not specified those properties are set programmatically with the default value.

-

- **Fault-tolerance properties**

  Note that those properties should not be altered if the programmer uses deployment descriptor files. See here [faultTolerance.xml] for more details.

  - **proactive.ft** . Two values are possible: **enable, disable**. If enable, the fault-tolerance is enable and a set of servers must be defined with the following properties. Default value is **disable**.
  - **proactive.ft.server.checkpoint** is the URL of the checkpoint server [faultTolerance.xml#configuration].

- **proactive.ft.server.location** is the URL of the location server [faultTolerance.xml#configuration].
- **proactive.ft.server.recovery** is the URL of the recovery process [faultTolerance.xml#configuration].
- **proactive.ft.server.resource** is the URL of the resource server [faultTolerance.xml#configuration].
- **proactive.ft.server.global** is the URL of the global server [faultTolerance.xml#configuration]. If this property is set, all others **proactive.fr.server.*** are ignored.
- **proactive.ft.ttc** is the value of the Time To Checkpoint [faultTolerance.xml#configuration] counter, in seconds. The default value is 30 sec.

- **Peer-to-Peer properties**

  - **proactive.p2p.acq** is the communication protocol that's used to communicate with this P2P Service. All ProActive communication protocols are supported: rmi, http, etc. Default is rmi.
  - **proactive.p2p.port** represents the port number on which to start the P2P Service. Default is 2410. The port is used by the communication protocol.
  - **proactive.p2p.noa: Number Of Acquaintances (NOA)** is the minimal number of peers one peer needs to know to keep up the infrastructure. By default, its value is 10 peers.
  - **proactive.p2p.ttu: Time To Update (TTU)** each peer sends an heart beat to its acquaintances. By default, its value is 1 minutes.
  - **proactive.p2p.ttl: Time To Live (TTL)** represents messages live time in hops of JVMs (node). By default, its value is 5 hops.
  - **proactive.p2p.msg_capacity** is the maximum memory size to stock message UUID. Default value is 1000 messages UUID.
  - **proactive.p2p.expl_msg** is the percentage of agree response when peer is looking for acquaintances. By default, its value is 66%.
  - **proactive.p2p.booking_max** uses during booking a shared node. It's the maximum time in millisecond to create at less an active object in the shared node. After this time and if no active objects are created the shared node will leave and the peer which gets this shared node will be not enable to use it more. Default is 3 minutes.
  - **proactive.p2p.nodes_acq_to** uses with descriptor file. It is the timeout in milliseconds for nodes acquisition. The default value is 3 minutes.
  - **proactive.p2p.lookup_freq** also uses with descriptor file. It is the lookup frequency in milliseconds for re-asking nodes. By default, it's value is 30 seconds.

- **proactive.p2p.multi_proc_nodes** if true deploying one shared nodes by CPU that means the p2p service which is running on a bi-pro will share 2 nodes, else only one node is shared independently of the number of CPU. By default, it's value is true, i.e. 1 shared node for 1 CPU.
- **proactive.p2p.xml_path** is the XML deployment descriptor file path for sharing nodes more than a single node.
-

-

- **rmi ssh properties**

The following properties are specific to the protocol rmissh [SSHTunneling.xml]

- **proactive.ssh.port** : the port number on which all the ssh daemons to which this JVM must connect to are expected to listen. If this property is not set, the default is 22.
- **proactive.ssh.username** : the username which will be used during authentication with all the ssh daemons to which this JVM will need to connect to. If this property is not set, the default is the user.name java property.
- **proactive.ssh.known_hosts** : a filename which identifies the file which contains the traditional ssh known_hosts list. This list of hosts is used during authentication with each ssh daemon to which this JVM will need to connect to. If the host key does not match the one stored in this file, the authentication will fail. If this property is not set, the default is System.getProperty ("user.home") + "/.ssh/known_hosts"
- **proactive.ssh.key_directory** : a directory which is expected to contain the pairs of public/private keys used during authentication. the private keys must not be encrypted. The public keys filenames must match "*.pub". Private keys are ignored if their associated public key is not present. If this property is not set, the default is System.getProperty ("user.home") + "/.ssh/"
- **proactive.tunneling.try_normal_first** : if this property is set to "yes", the tunneling code always attempts to make a direct rmi connection to the remote object before tunneling. If this property is not set, the default is not to make these direct-connection attempts. This property is especially useful if you want to deploy a number of objects on a LAN where only one of the hosts needs to run with the rmissh protocol to allow hosts outside the LAN to connect to this frontend host. The other hosts located on the LAN can use the try_normal_first property to avoid using tunneling to make requests to the LAN frontend.
- **proactive.tunneling.connect_timeout** : this property specifies how long the tunneling code will wait while trying to establish a connection to a remote host before declaring that the connection failed. If this property is not set, the default value is 2000ms.
- **proactive.tunneling.use_gc** : if this property is set to "yes", the client JVM does not destroy the ssh tunnels are soon as they are not used anymore. They are queued into a list of unused tunnels which can be reused. If this property is not set or is set to another value, the tunnels are destroyed as soon as they are not needed anymore by the JVM.
- **proactive.tunneling.gc_period** : this property specifies how long the tunnel garbage collector will wait before destroying a unused tunnel. If a tunnel is older than this value, it is automatically destroyed. If this property is not set, the default value is 10000ms.

-

- **Other properties**

  - **proactive.rmi.port** represents the port number on which to start the RMIRegistry. Default is 1099. If an RMIRegistry is already running on the given port, jms use the existing registry
  - **proactive.http.port** represents the port number on which to start the HTTP server. Default is 2010. If this port is occupied by another application, the http server starts on the first free port(given port is incremented transparently)
  - **proactive.useIPaddress** if set to **true**, IP adresses will be used instead of machines names. This property is particularly usefull to deal with sites that do not host a DNS
  - **proactive.hostname** when this property is set, the host name on which the jvm is started is given by the value of the property. This property is particularly usefull to deal with machines with two network interfaces
  - **proactive.locationserver** represents the location server class to instantiate when using Active Objects with Location Server
  - **proactive.locationserver.rmi** represents the url under which the Location Server is registered in the RMIRegistry
  - **fractal.provider** This property defines the bootstrap component for the Fractal component model
  - **proactive.classloader** runtimes created with this property enabled fetch missing classes using a special mechanism [../core/classloader/package.xml]. This is an alternative to RMI dynamic class downloading, useful for instance when performing hierarchical deployment.
  -

  Note that as mentionned above, user-defined properties can be added.

9.5. Example

Configuration file must have following structure:

```
<ProActiveUserProperties>
 <properties>
 <prop
key="schema.validation"
value="disable"/>
 <prop
key="proactive.future.ac"
value="enable"/>
 <prop
key="proactive.communication.protocol"
value="rmi"/>
 <prop
key="proactive.rmi.port"
value="2001-2005"/>
 ....
 <prop
key="myprop"
value="myvalue"/>
 ....
```

```
 </properties>
</ProActiveUserProperties>
```

Note that tag **ProActiveUserProperties**, tag **properties** and the model: <prop **key**="somekey" **value**="somevalue"/> are mandatory for ProActive to parse correctly the document

Chapter 10. Variable Contracts for Descriptors

10.1. Variable Contracts for Descriptors

10.1.1. Principle

The objective of this feature is to allow the use of variables with XML descriptors. Variables can be defined: directly in the descriptor, using independent files, or inside the deploying application's code (with an API).

The variable tags are usefull inside a descriptor because they can factorize frequent parameters. (For example, a variable like ${PROACTIVE_HOME} can be defined, set and used in an XML Descriptor.) But also, because they can be used to establish a contract between the Program and the Descriptor.

10.1.2. Variable Types

Type	Ability to set value	Ability to set empty value	Priority
DescriptorVariable	Descriptor	Program	Descriptor
ProgramVariable	Program	Descriptor	Program
DescriptorDefaultVariable	Descriptor, Program	--	Program
ProgramDefaultVariable	Program, Descriptor	--	Descriptor
JavaPropertyVariable	Descriptor, Program	--	-

Variables can be set in more than one place. When the value is set on multiple places, then the definition specified in the priority column will take precedence.

10.1.3. Variable Types User Guide

To help identify the user cases where the variable types might be useful, we have defined the concept of programmer and deployer. The programmer is the person writing the application code. The deployer corresponds

to the responsible of writing the deployment descriptor. The variables represent rights and responsabilities between the two parties (contract) as specified in the following table:

Variable Contracts for Descriptors

Type	Behavior	When to use this type
DescriptorVariable	The value has to be set in the descriptor, and cannot be specified in the program.	If the deployer wants to use a value, without giving the possibility to the programmer to modify it. The programmer can define this variable to empty, to force the descriptor to set a value.
ProgramVariable	The value must be set in the program, and cannot be specified in the descriptor.	If the programmer wants to use a value, without giving the possibility to the descriptor to modify it. The descriptor can define this variable to empty, to force the programmer to set a value.
DescriptorDefaultVariable	A default value must be specified in the descriptor. The programmer has the ability not to change the value in the program. Nevertheless, if the value is changed in the program, then this new value will have precedence over the one defined in the descriptor.	If the programmer may override the default value, but the responsability of setting a default belongs to the deployer.
ProgramDefaultVariable	A default value must be specified in the program. The descriptor has the ability not to change the value. Nevertheless, if the	If the deployer may override the default value, but the responsability of setting a default belongs to the programmer.

-->

10.1.4. Variables Example

10.1.4.1. Descriptor Variables

All variables must be set in a variable section at the beginning of the descriptor file in the following way:

```
<variables>
< DescriptorVariable name="
PROACTIVE_HOME" value="
ProActive/dist/ProActive"/>
< DescriptorDefaultVariable name="
NUMBER_OF_VIRTUAL_NODES" value="
4"/>
< ProgramVariable name="
VIRTUAL_NODE_NAME" value=""/>

<!-- Example Using java properties -->
< JavaPropertyVariable name="
USER_HOME" value="
java.home"/>

<!-- Include external variables from files-->
< IncludeXMLFile location="
file.xml"/>
< IncludePropertyFile location="
file.properties"/>
</variables>

...
<!-- Usage example-->
<classpath>
<absolutePath value="
${USER_HOME}/${PROACTIVE_HOME}/ProActive.jar"/>
...
</classpath>
...
```

10.1.4.2. Program Variables

```
VariableContract variableContract= new VariableContract();

variableContract.setVariableFromProgram( "
VIRTUAL_NODE_NAME", "
testnode",
VariableContractType.ProgramVariable);
variableContract.setVariableFromProgram( "
NUMBER_OF_VIRTUAL_NODES", "
10",
VariableContractType.DescriptorDefaultVariable);

ProActiveDescriptor pad =↩
ProActive.getProactiveDescriptor(XML_LOCATION, va\
```

```
riableContract);
```

10.1.5. Variable Types User Guide

To help identify the user cases where the variable types might be useful, we have defined the concept of programmer and deployer. The programmer is the person writing the application code. The deployer corresponds to the responsible of writing the deployment descriptor. The variables represent rights and responsabilities between the two parties (contract) as specified in the following table:

Variable Contracts for Descriptors

Type	Behavior	When to use this type
DescriptorVariable	The value has to be set in the descriptor, and cannot be specified in the program.	If the deployer wants to use a value, without giving the possibility to the programmer to modify it. The programmer can define this variable to empty, to force the descriptor to set a value.
ProgramVariable	The value must be set in the program, and cannot be specified in the descriptor.	If the programmer wants to use a value, without giving the possibility to the descriptor to modify it. The descriptor can define this variable to empty, to force the programmer to set a value.
DescriptorDefaultVariable	A default value must be specified in the descriptor. The programmer has the ability not to change the value in the program. Nevertheless, if the value is changed in the program, then this new value will have precedence over the one defined in the descriptor.	If the programmer may override the default value, but the responsability of setting a default belongs to the deployer.
ProgramDefaultVariable	A default value must be specified in the program. The descriptor has the ability not to change the value. Nevertheless, if the	If the deployer may override the default value, but the responsability of setting a default belongs to the programmer.

-->

10.1.6. Variables Example

10.1.6.1. Descriptor Variables

All variables must be set in a variable section at the beginning of the descriptor file in the following way:

```
<variables>
< DescriptorVariable name="
PROACTIVE_HOME" value="
ProActive/dist/ProActive"/>
< DescriptorDefaultVariable name="
NUMBER_OF_VIRTUAL_NODES" value="
4"/>
< ProgramVariable name="
VIRTUAL_NODE_NAME" value=""/>


<!-- Example Using java properties -->

< JavaPropertyVariable name="
USER_HOME" value="
java.home"/>


<!-- Include external variables from files-->

< IncludeXMLFile location="
file.xml"/>
< IncludePropertyFile location="
file.properties"/>
</variables>

...

<!-- Usage example-->

<classpath>
<absolutePath value="
${USER_HOME}/${PROACTIVE_HOME}/ProActive.jar"/>
...
</classpath>
...
```

10.1.6.2. Program Variables

```
VariableContract variableContract= new VariableContract();

variableContract.setVariableFromProgram( "
VIRTUAL_NODE_NAME", "
testnode",
VariableContractType.ProgramVariable);
variableContract.setVariableFromProgram( "
```

```
      NUMBER_OF_VIRTUAL_NODES", "
      10",
      VariableContractType.DescriptorDefaultVariable);

      ProActiveDescriptor pad =
ProActive.getProactiveDescriptor(XML_LOCATION, va\
      riableContract);
```

10.1.7. External Variable Definitions Files

10.1.7.1. XML Files

Is built using XML property tags.

File : file.xml

```
<!-- Definition of the specific context -->
<variables>
< DescriptorVariable name="
USER_HOME" value="
/usr/home/team"/>
< DescriptorVariable name="
PROACTIVE_HOME" value="
ProActive/dist/ProActive"/>
< DescriptorVariable name="
NUM_NODES" value="
45"/>
</variables>
```

10.1.7.2. Properties Files

This approach uses Sun microsystems properties file format [http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.xml#load(jav
The format is plain text with one definition per line in the format **variable = value**, as shown in the following example:

File : file.properties

```
# Definition of the specific context
USER_HOME = /usr/home/team
PROACTIVE_HOME = ProActive/dist/ProActive
NUM_NODES : 45
```

Variables defined in this format will be declared as **DescriptorVariable** type. Note that colon (:) can be used instead of equal (=).

10.1.8. Program Variable API

10.1.8.1. Relevant import packages

```
import org.objectweb.proactive.core.xml.VariableContract;
import org.objectweb.proactive.core.xml.VariableContractType;
```

10.1.8.2. Available Variable Types

- VariableContractType.**DefaultVariable**
- VariableContractType.**DescriptorDefaultVariable**
- VariableContractType.**ProgramVariable**
- VariableContractType.**ProgramDefaultVariable**
- VariableContractType.**JavaPropertyVariable**

10.1.8.3. API

The API for setting variables from the Program is shown below. The **name** corresponds to the variable name, and the **value** to the variable content. The **type** corresponds to a VariableContractType.

```
public void VariableContract.setVariableFromProgram( String
name, String
value, VariableContractType
type);
public void VariableContract.setVariableFromProgram( HashMap
map, VariableContractType
type);
```

The API for adding a multiple variables is shown above. The variable **name**/**value** pair is specified as the key/content of the HashMap.
-->

10.1.9. External Variable Definitions Files

10.1.9.1. XML Files

Is built using XML property tags.

File : file.xml

```
<!-- Definition of the specific context -->
<variables>
< DescriptorVariable name="
USER_HOME" value="
/usr/home/team"/>
< DescriptorVariable name="
PROACTIVE_HOME" value="
ProActive/dist/ProActive"/>
< DescriptorVariable name="
NUM_NODES" value="
45"/>
```

```
</variables>
```

10.1.9.2. Properties Files

This approach uses Sun microsystems properties file format [http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.xml#load(jav
The format is plain text with one definition per line in the format **variable = value**, as shown in the following example:

File : file.properties

```
# Definition of the specific context
USER_HOME = /usr/home/team
PROACTIVE_HOME = ProActive/dist/ProActive
NUM_NODES : 45
```

Variables defined in this format will be declared as **DescriptorVariable** type. Note that colon (:) can be used instead of equal (=).

10.1.10. Program Variable API

10.1.10.1. Relevant import packages

```
import org.objectweb.proactive.core.xml.VariableContract;
import org.objectweb.proactive.core.xml.VariableContractType;
```

10.1.10.2. Available Variable Types

- VariableContractType.**DefaultVariable**
- VariableContractType.**DescriptorDefaultVariable**
- VariableContractType.**ProgramVariable**
- VariableContractType.**ProgramDefaultVariable**
- VariableContractType.**JavaPropertyVariable**

10.1.10.3. API

The API for setting variables from the Program is shown below. The **name** corresponds to the variable name, and the **value** to the variable content. The **type** corresponds to a VariableContractType.

```
public void VariableContract.setVariableFromProgram( String
name, String
value, VariableContractType
type);
public void VariableContract.setVariableFromProgram( HashMap
map, VariableContractType
type);
```

The API for adding a multiple variables is shown above. The variable **name**/**value** pair is specified as the key/content of the HashMap.

Chapter 11. ProActive File Transfer Model

The following document describes our current proposal model for supporting file transfer in ProActive.

11.1. Introduction and Concepts

We believe it is interesting for us, and the ProActive users, to provide support for FileTransfer

Currently we recognize the following type of files:

- 1. ProActive Deployment Required Files (jars, policy, log4, etc)
- 2. User Application Data Files (User application input)
- 3. User Application Code Files (jar, classes or native code)
- 4. JDK files (the jvm runtime and libraries)

Also, this files can be transfered:

- 1. To a remote location (Deploy)
- 2. From a remote location (Retrieve)

The FileTransfer could happen:

- 1. Before deployment.
- 2. After deployment but before the user launches his/her application.
- 3. During user application.
- 4. After the user application has finished.

11.2. Objectives

11.2.1. Main objective

- Support File Transfer in ProActive

11.2.2. Specific Objectives

- Specify file transfer in descriptor files.
- If the process supports FileTransfer use the protocol support (processDefault), or some other protocol.
- Else, use FileTransfer as file streaming in ProActive Runtime (jvm2jvm)

11.3. Supported Protocols

- scp
- rcp
- unicore (Unicore processDefault)
- nordugrid (Nordugrid processDefault)

11.4. FileTransfer Design

FileTransfer definitions are divided in two:

11.4.1. Abstract Definition

This definitions can be referenced from a VirtualNode. They contain the most basic information of a FileTransfer:

- A unique definition identification name.
- Files: source and optionally the destination name.
- Directories: source and optionally the destination name. Also the exclude and include patterns (not yet available feature).

References from the VirtualNode are made using the unique definition name.

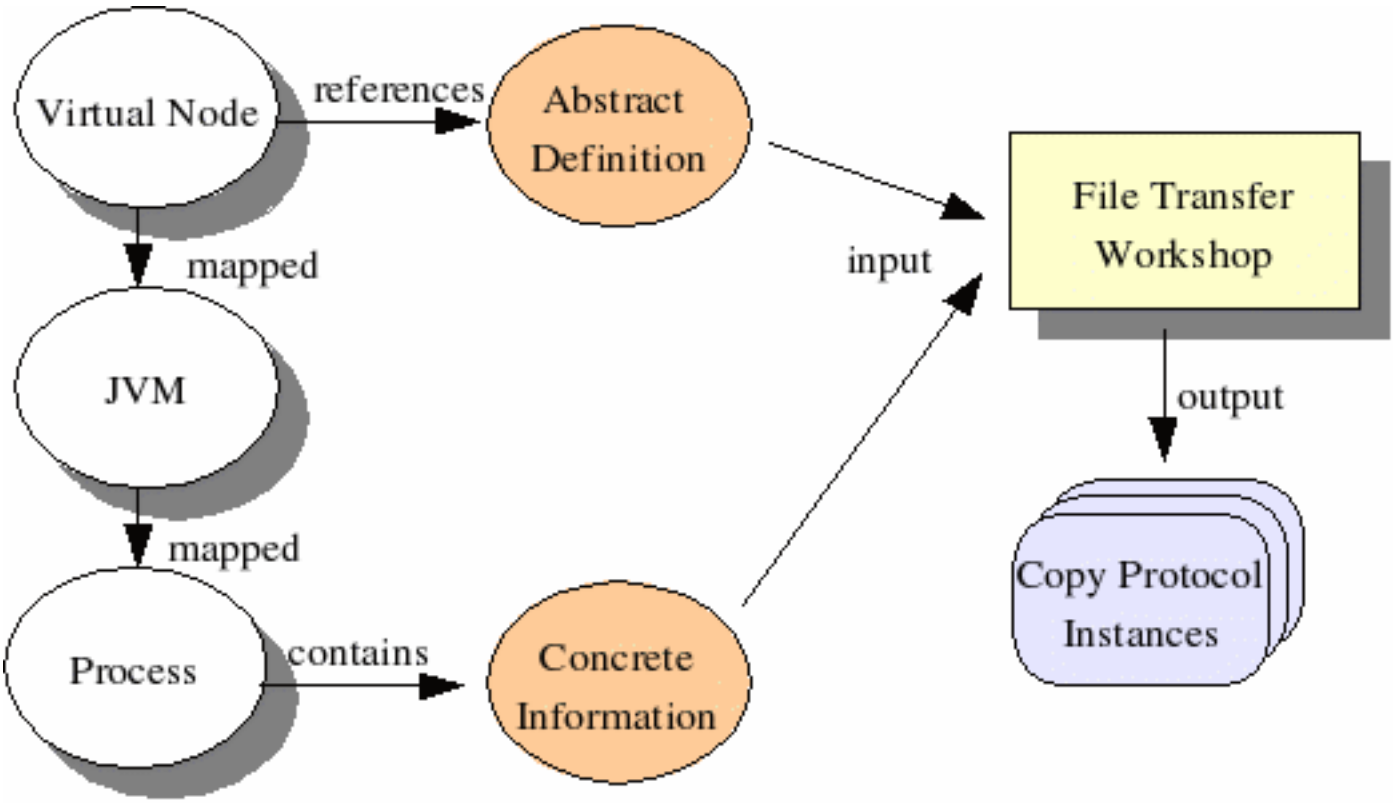
11.4.2. Concrete Definition

These definitions contain more architecture specific information, and are therefore contained within the Process:

- A reference to an abstract definition, or the "implicit" indicating the reference will be inherited from the VirtualNode.
- A sequence of Copy Protocols that will be used.
- Source and Destination information: prefix, username, hostname, file separator, etc...

If some of this information (like username or hostname) can be inferred from the process, it is not necessary to declare it in the definition. Optionally, the information contained in the protocol can be overridden if specified.

11.4.3. How it works



When a FileTransfer starts, both abstract and concrete information are merged using the FileTransfer Workshop. The result of this process correspons to a sequence of CopyProtocols, as specified in the Concrete Definition.

Each CopyProtocol will be tried before the deployment takes place, until one succeeds. After one succeed are all fail, the process deployment will take place.

11.5. Descriptor FileTransfer XML Tags

The abstract concept: "FileTransfer" is introduced. This concept is independent off the actual process used, but also flexible and configurable at the process level. To define this abstract representation of the FileTransfer:

```
....
</deployment>
<
FileTransferDefinitions>
  <
FileTransfer id="
example">
    <
file src="hello.dat" dest="world.dat"/>
    <
file src="hello.jar" dest="world.jar"/>
    <
file src="hello.class" dest="world.class"/>
    <
dir src="exampledir" dest="exampledir"/>
 </
FileTransfer>
 ...
```

```
</
FileTransferDefinitions>
<infrastructure>
....
```

Note: exclude and include are not yet supported features

This tags can be referenced at the the VirtualNode level and the Process level.

- **VirtualNode**

```
<VirtualNode name="exampleVNode"
FileTransferDeploy="
example"/>
```

- **Process**

```
<processDefinition id="xyz">
 <sshProcess>...

<!-- Inside the process, the FileTransfer tag becomes an element instead
of \
an attribute.
        This happens because FileTransfer information is process specif\
ic.
        Note that the destination hostname and username can be omitted,\
 and implicitly inferred
        from the process information. -->

   <
FileTransferDeploy="
implicit">
<!-- referenceID or keyword "implicit" (inherit)-->

    <
copyProtocol>processDefault, scp, rcp</
copyProtocol>
    <
sourceInfo prefix="/home/user"/>
    <
destinationInfo prefix="/tmp" hostname="foo.org" username="smith" />
   </
FileTransferDeploy>
 </sshProcess>
</processDefinition>
```

The **implicit** keyword means the **FileTransferDeploy** identifier is inherited from the VN definition. If hostname and username are implicitly defined in the process tag, then this information will be used. But, if the user desires to override this information he can specify it as an attribute in the **destinationInfo** tag. When coallocating more than one VN on a JVM, and therefore on the same process, all files inherited from the multiple VN definitions will be transfered.

Note: FileTransferRetrieve is not yet supported.

Chapter 12. Using SSH tunneling for RMI or HTTP communications

12.1. Overview

ProActive allows users to **tunnel** all of their RMI or HTTP communications over **SSH**: it is possible to specify in ProActive deployment descriptors which JVMs should **export** their RMI objects through a SSH tunnel.

This kind of feature is useful for two reasons:

- it might be necessary to encrypt the RMI communications to improve the RMI security model.
- the configuration of the network in which a given ProActive application is deployed might contain firewalls which reject or drop direct TCP connections to the machines which host RMI objects. If these machines are allowed to receive ssh conections over their port 22 (or another port number), it is possible to multiplex and demultiplex all RMI connections to that host through its ssh port.

To successfully use this feature with reasonable performance, it is **mandatory** to understand:

- the configuration of the underlying network: **location and configuration of the firewalls**.
- the communication patterns of the underlying ProActive runtime: **which JVM makes requests to which JVMs.**
- the communication patterns of your ProActive objects: **which object makes requests to which object**. For example: A -> B, B -> C, A ->C

12.2. Configuration of the network

No two networks are alike. The only thing they share is the fact that they are all different. Usually, what you must look for is:

- is A **allowed** to open a connection to B ?
- is B **allowed** to open a connection to A ? (networks are rarely symetric)

If you use a TCP or a UDP-based communication protocol (ie: RMI is based on TCP), these questions can be translated into "what **ports** on B is A **allowed** to open a connection to ?". Once you have answered this question for all the hosts used by your application, write down a small diagram which outlines what kind of connection is possible. For example:

This diagram summarizes the fact that host A is protected by a firewall which allows outgoing connections without control but allows only incoming connections on port 22. Host B is also protected by a similar firewall.

12.3. ProActive runtime communication patterns

To execute a ProActive application, you need to **"deploy"** it. Deployment is performed by the ProActive runtime and is configured by the ProActive deployment descriptor of the initial host. During deployment, each newly-created ProActive runtime performs a request to the initial ProActive runtime. The initial runtime also performs at least one request on each of these distant runtime.

This 2-way communication handshake makes it necessary to **correctly configure the network** to make sure that the filtering described above does not interfere with the normal operation of the ProActive runtimes.

12.4. ProActive application communication patterns.

Once an application is properly deployed, the application objects deployed by the ProActive runtime start making requests to each other. It is important to properly identify what object connects to what object to identify the influence of the network configuration on these communication patterms.

12.5. ProActive communication protocols

Whenever a request is made to a non-local ProActive object, this request is performed with the communication protocol specified by the destination JVM. Namely, each JVM is characterized by a a unique property named **proactive.communication.protocol** which is set to one of:

- rmi
- http
- rmissh
- ibis
- jini

This property uniquely identifies the protocol which is used by each client of the JVM to send data to this JVM. To use different protocols for different JVMs, two solutions exist:

- one is to edit the **ProActive deployment descriptors** and to pass the property as a command-line option to the JVM:

-

```
<jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
....
<jvmParameters>
   <parameter value="-Dproactive.communication.protocol=rmissh"/>
</jvmParameters>
...
</jvmProcess>
```

- the other one is to set in the **ProActive Configuration file**(introduced in a previous chapter) on the remote host the property proactive.communication.protocol to the desired protocol

```
<prop key="proactive.communication.protocol" value="rmissh"/>
```

Finally, if you want to set this property on the **initial** deployment JVM (the JVM that starts the application), you will need to specify the -Dproactive.communication.protocol=rmissh argument yourself on the JVM command line.

12.6. The rmissh communication protocol.

This protocol is a bit special because it keeps a lot of compatibility with the rmi protocol and a lot of options are available to "**optimize**" it.

This protocol can be used to automatically **tunnel** all RMI communications through SSH tunnels. Whenever a client wishes to access a distant rmissh server, rather than connecting directly to the distant server, it first creates a SSH tunnel (so-called port-forwarding) from a random port locally to the distant server on the distant host/port. Then, all it has to do to connect to this server is to pretend this server is listening on the local random port choosen by the ssh tunnel. The ssh daemon running on the server host receives the data for this tunnel, decapsulates it and forwards it to the real server.

Thus, whenever you request that a JVM be accessed only through rmissh (namely, whenever you set its **proactive.communication.protocol to rmissh**), you need to make sure that an ssh daemon is running on its host. ProActive uses the **jsch** client ssh library to connect to this daemon.

The properties you can set to configure the behavior of the ssh tunneling code are listed below. All these properties are client-side properties:

- **proactive.ssh.port** : the port number on which all the ssh daemons to which this JVM must connect to are expected to listen. If this property is not set, the default is 22.
- **proactive.ssh.username** : Two possible syntaxes: username alone .e.g. **proactive.ssh.username=jsmith**, it represents the username which will be used during authentication with all the ssh daemons to which this JVM will need to connect to.

  Or you can use the form **proactive.ssh.username=username1@machine1;username2@machine2;...;usernameN@machineN**
  Note that several usernames without machine's names is not allowed and won't be parsed properly.

  If this property is not set, the default is the user.name java property.

- **proactive.ssh.known_hosts** : a filename which identifies the file which contains the traditional ssh known_hosts list. This list of hosts is used during authentication with each ssh daemon to which this JVM will need to connect to. If the host key does not match the one stored in this file, the authentication will fail. If this property is not set, the default is System.getProperty ("user.home") + "/.ssh/known_hosts"

- **proactive.ssh.key_directory** : a directory which is expected to contain the pairs of public/private keys used during authentication. the private keys must not be encrypted. The public keys filenames must match "*.pub". Private keys are ignored if their associated public key is not present. If this property is not set, the default is System.getProperty ("user.home") + "/.ssh/"

- **proactive.tunneling.try_normal_first** : if this property is set to "yes", the tunneling code always attempts to make a direct rmi connection to the remote object before tunneling. If this property is not set, the default is not to make these direct-connection attempts. This property is especially useful if you want to deploy a number of objects on a LAN where only one of the hosts needs to run with the rmissh protocol to allow hosts outside the LAN to connect to this frontend host. The other hosts located on the LAN can use the try_normal_first property to avoid using tunneling to make requests to the LAN frontend.

- **proactive.tunneling.connect_timeout** : this property specifies how long the tunneling code will wait while trying to establish a connection to a remote host before declaring that the connection failed. If this property is not set, the default value is 2000ms.

- **proactive.tunneling.use_gc** : if this property is set to "yes", the client JVM does not destroy the ssh tunnels as soon as they are not used anymore. They are queued into a list of unused tunnels which can be reused. If this property is not set or is set to another value, the tunnels are destroyed as soon as they are not needed anymore by the JVM.

- **proactive.tunneling.gc_period** : this property specifies how long the tunnel garbage collector will wait before destroying a unused tunnel. If a tunnel is older than this value, it is automatically destroyed. If this property is not set, the default value is 10000ms.

Note that the use of SSH tunneling over RMI still allows dynamic classloading through HTTP. For the dynamic classloading our protocol creates an SSH tunnel over HTTP, in order to get missing classes. It is also important to notice that all you have to do in order to use SSH tunneling is to set the **proactive.communiaction.protocol** property to **rmissh** and to use the related properties if needed(in major cases default behavior is sufficient), ProActive takes care of everything else.

Chapter 13. Fault-Tolerance

13.1. Overview

ProActive can provide fault-tolerance capabilities through two differents protocols: a Communication-Induced Checkpointing protocol (CIC) or a pessimistic message logging protocol (PML). Making a ProActive application fault-tolerant is **fully transparent**; active objects are turned fault-tolerant using Java properties that can be set in the deployment descriptor [Descriptor.xml] . The programmer can select *at deployment time* the most adapted protocol regarding the application and the execution environment.

Persistence of active objects is obtained through standard Java serialization; a checkpoint thus consists in an object containing a serialized copy of an active object and few informations related to the protocol. As a consequence, a fault-tolerant active object must be serializable.

13.1.1. Communication Induced Checkpointing (CIC)

Each active object in a CIC fault-tolerant application have to checkpoint at least every **TTC** (Time To Checkpoint) seconds. When all the active objects have taken a checkpoint, a **global state** is formed. If a failure occurs, the *entire* application must restarts from such a global state. The TTC value depends mainly on the assessed frequency of failures. A little TTC value leads to very frequent global state creation and thus to a little rollback in the execution in case of failure. But a little TTC value leads also to a bigger overhead between a non-fault-tolerant and a fault-tolerant execution. The TTC value can be set by the programmer in the deployment descriptor.

The failure-free overhead induced by the CIC protocol is usually low, and this overhead is quasi-independent from the message communication rate. The counterpart is that the recovery time could be long since all the application must restart after the failure of one or more active object.

13.1.2. Pessimistic message logging (PML)

Each active object in a PML fault-tolerant application have to checkpoint at least every TTC seconds and all the messages delivered to an active object are logged on a stable storage. There is no need for global synchronization as with CIC protocol, each checkpoint is independent: if a failure occurs, only the faulty process have to recover from its latest checkpoint. As for CIC protocol, the TTC value impact the global failure-free overhead, but the overhead is more linked to the communication rate of the application.

Regarding the CIC protocol, the PML protocol induces a higher overhead on failure-free execution, but the recovery time is lower as a single failure does not involve all the system.

Warning: For the version 3.0, those two protocols are not compatible: a fault-tolerance application can use **only** one of the two protocols. This compatibility will be provide in the next version.

13.2. Making a ProActive application fault-tolerant

13.2.1. Resource Server

To be able to recover a failed active object, the fault-tolerance system must have access to a *resource server*. A resource server is able to return a free node that can host the recovered active object.

A resource server is implemented in ProActive in `ft.servers.resource.ResourceServer`. This server can store free nodes by two differents way:

- at deployment time: the user can specify in the deployment descriptor a resource virtual node. Each node mapped on this virtual node will automaticaly register itself as free node at the specified resource server.
- at execution time: the resource server can use an underlying p2p network [p2p.xml] to reclaim free nodes when a hosting node is needed.

Note that those two mechanisms can be combined. In that case, the resource server first provides node registered at deployment time, and when no more such nodes are available, the p2p network is used.

13.2.2. Fault-Tolerance servers

Fault-tolerance mechanism needs servers for the checkpoints storage, the localization of the active objects, and the failure detection. Those servers are implemented in the current version as a unique server (`ft.servers.FTServer`), that implements the interfaces of each server (`ft.servers.*.*`). This global server also includes a resource server.

This server is a classfile server for recovered active objects. It must thus have access to all classes of the application, i.e. it must be started with **all classes of the application in its classpath**.

The global fault-tolerance server can be launched using the `ProActive/scripts/[unix|windows]/FT/startGlobalFT` script, with 5 optional parameters:

- the protocol: `-proto [cic|pml]`. Default value is `cic`.
- the server name: `-name <serverName>`. The default name is FTServer.
- the port number: `-port <portNumber>`. The default port number is 1100.
- the fault detection period: `-fdperiod <periodInSec>`. This value defines the time between two consecutive fault detection scanning. The default value is 10 sec. Note that an active object is considered as faulty when it becomes unreachable, i.e. when it becomes unable to receive a message from another active object.
- the URL of a p2p service [p2p.xml] that can be used by the resource server: `-p2p <serviceURL>`. There is no default value for this option.

The server can also be directly launched in the java source code, using `org.objectweb.proactive.core.process.JVMPr` class:

```
GlobalFTServer server = new JVMProcessImpl(new↩
org.objectweb.proactive.core\
.process.AbstractExternalProcess.StandardOutputMessageLogger());

this.server.setClassname("org.objectweb.proactive.core.body.ft.servers.Start\

FTServer");
this.server.startProcess();
```

Note that if one of the servers is unreachable when a fault-tolerant application is deploying, fault-tolerance is automatically and transparently disabled for all the application.

13.2.3. Configure fault-tolerance for a ProActive application

Fault-tolerance capabilities of a ProActive application are set in the deployment descriptor, using the `faultTolerance` service. This service is attached to *a virtual node*: active objects that are deployed on this virtual node are turned fault-tolerant. This service must first defines the protocol that have to be used for this application. The user can select the appropriate protocol with the entry `<protocol type="[cic|pml]"/>` in the definition of the service.

The service also defines **servers URLs** :

- `<globalServer url="..."/>` set the URL of a *global* server, i.e. a server that implements all needed methods for fault-tolerance mechanism (stable storage, fault detection, localization). If this value is set, all others URLs will be *ignored*.
- `<checkpointServer url="..."/>` set the URL of the checkpoint server, i.e. the server where checkpoints are stored.
- `<locationServer url="..."/>` set the URL of the location server, i.e. the server responsible for giving references on failed and recovered active objects.
- `<recoveryProcess url="..."/>` set the URL of the recovery process, i.e. the process responsible for launching the recovery of the application after a failure.
- `<resourceServer url="..."/>` set the URL of the resource server, i.e. the server responsible for providing free nodes that can host a recovered active object.

Finally, the **TTC** value is set in fault-tolerance service, using `<ttc value="x"/>`, where x is expressed in *seconds*. If not, the default value (30 sec) is used.

13.2.4. A deployment descriptor example

Here is an example of deployment descriptor that deploys 3 virtual nodes : one for deploying fault-tolerant active objects, one for deploying non-fault-tolerant active object (if needed), and one as resource for recovery. The two fault-tolerance behaviors correspond to two fault-tolerance services, `appli` and `resource`. Note that non-fault-tolerant active objects can communicate with fault-tolerant active objects as usual. Chosen protocol is CIC and TTC is set to 5 sec for all the application.

```
<ProActiveDescriptor>
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="NonFT-Workers" property="multiple"/>
   <virtualNode name="FT-Workers" property="multiple" ftServiceId="appli"/>
   <virtualNode name="Failed" property="multiple" ftServiceId="resource"/>
  </virtualNodesDefinition>
 </componentDefinition>
 <deployment>
  <mapping>
   <map virtualNode="NonFT-Workers">
    <jvmSet>
     <vmName value="Jvm1"/>
    </jvmSet>
   </map>
   <map virtualNode="FT-Workers">
    <jvmSet>
     <vmName value="Jvm2"/>
    </jvmSet>
   </map>
   <map virtualNode="Failed">
    <jvmSet>
```

```
      <vmName value="JvmS1"/>
      <vmName value="JvmS2"/>
     </jvmSet>
    </map>
   </mapping>
   <jvms>
    <jvm name="Jvm1">
     <creation>
      <processReference refid="linuxJVM"/>
     </creation>
    </jvm>
    <jvm name="Jvm2">
     <creation>
      <processReference refid="linuxJVM"/>
     </creation>
    </jvm>
    <jvm name="JvmS1">
     <creation>
      <processReference refid="linuxJVM"/>
     </creation>
    </jvm>
    <jvm name="JvmS2">
     <creation>
      <processReference refid="linuxJVM"/>
     </creation>
    </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="linuxJVM">
    <jvmProcess
 class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
  </processes>
  <services>
   <serviceDefinition id="appli">
    <faultTolerance>
   <protocol type="cic"></protocol>
       <globalServer url="rmi://localhost:1100/FTServer"></globalServer>
     <ttc value="5"></ttc>
    </faultTolerance>
   </serviceDefinition>
   <serviceDefinition id="resource">
    <faultTolerance>
   <protocol type="cic"></protocol>
       <globalServer  url="rmi://localhost:1100/FTServer"></globalServer>
       <resourceServer url="rmi://localhost:1100/FTServer"></resourceServer>
     <ttc value="5"></ttc>
    </faultTolerance>
   </serviceDefinition>
  </services>
 </infrastructure>
</ProActiveDescriptor>
```

13.3. Programming rules

13.3.1. Serializable

Persistence of active objects is obtained through standard Java serialization; a checkpoint thus consists in an object containing a serialized copy of an active object and a few informations related to the protocol. As a consequence, a fault-tolerant active object *must be serializable*. If a non serializable object is activated on a fault-tolerant virtual node, fault-tolerance is automatically and transparently disabled for this active object.

13.3.2. Standard Java main method

Standard Java thread, typically main method, cannot be turned fault-tolerant. As a consequence, if a standard main method interacts with active objects during the execution, consistency after a failure can no more be ensured: after a failure, all the active objects will roll back to the most recent global state *but the main will not*.

So as to avoid such inconsistency on recovery, the programmer must minimizes the use of standard main by, for example, delegating the initialization and launching procedure to an active object.

```
...
public static void main(String[] args){
  Initializer init =↩
(Initializer)(ProActive.newActive("Initializer.getClas\
s.getName()", args);
  init.launchApplication();
  System.out.println("End of main thread");
}
...
```

The object `init` is an active object, and as such will be rolled back if a failure occurs: the application is kept consistent.

13.3.3. Checkpointing occurrence

To keep fault-tolerance fully transparent (see the technical report [http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5
for more details), active objects can take a checkpoint *before the service of a request*. As a first consequence, if the service of a request is infinite, or at least much greater than TTC, the active object that serves such a request can no more take checkpoints. If a failure occurs during the execution, this object will force the entire application to rolls back to the beginning of the execution. The programmer must thus avoid infinite method such as

```
...
public void infiniteMethod(){
  while (true){
    this.doStuff();
  }
}
...
```

The second consequence concerns the definition of the `runActivity()` method (see `runActive` [http://www-sop.inria.fr/oasis/ProActive/doc/api/org/objectweb/proactive/RunActive.xml]). Let us consider the following example :

```
...
public void runActivity(Body body) {
  org.objectweb.proactive.Service service = new
org.objectweb.proactive.Se\
rvice(body);
  while (body.isActive()) {
    Request r = service.blockingRemoveOldest();
    ...
    /* CODE A */
    ...
    /* CHECKPOINT OCCURRENCE */
    service.serve(r);
  }
}
...
```

If a checkpoint is triggered before the service of `r`, it characterizes the state of the active object at the point `/* CHECKPOINT OCCURRENCE */`. If a failure occurs, this active object is restarted by calling the `runActivity()` method, *from a state in which the code `/* CODE A */` has been already executed*. As a consequence, the execution looks like if `/* CODE A */` was executed two times.

The programmer should then avoid to alter the state of an active object in the code preceding the call to `service.serve(r)` when he redefines the `runActivity()` method.

13.3.4. Activity Determinism

All the activities of a fault-tolerant application must be deterministic (see the technical report [http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5246.pdf] for more details). The programmer must then avoid the use of non-deterministic methods such as `Math.random()`.

13.3.5. Limitations

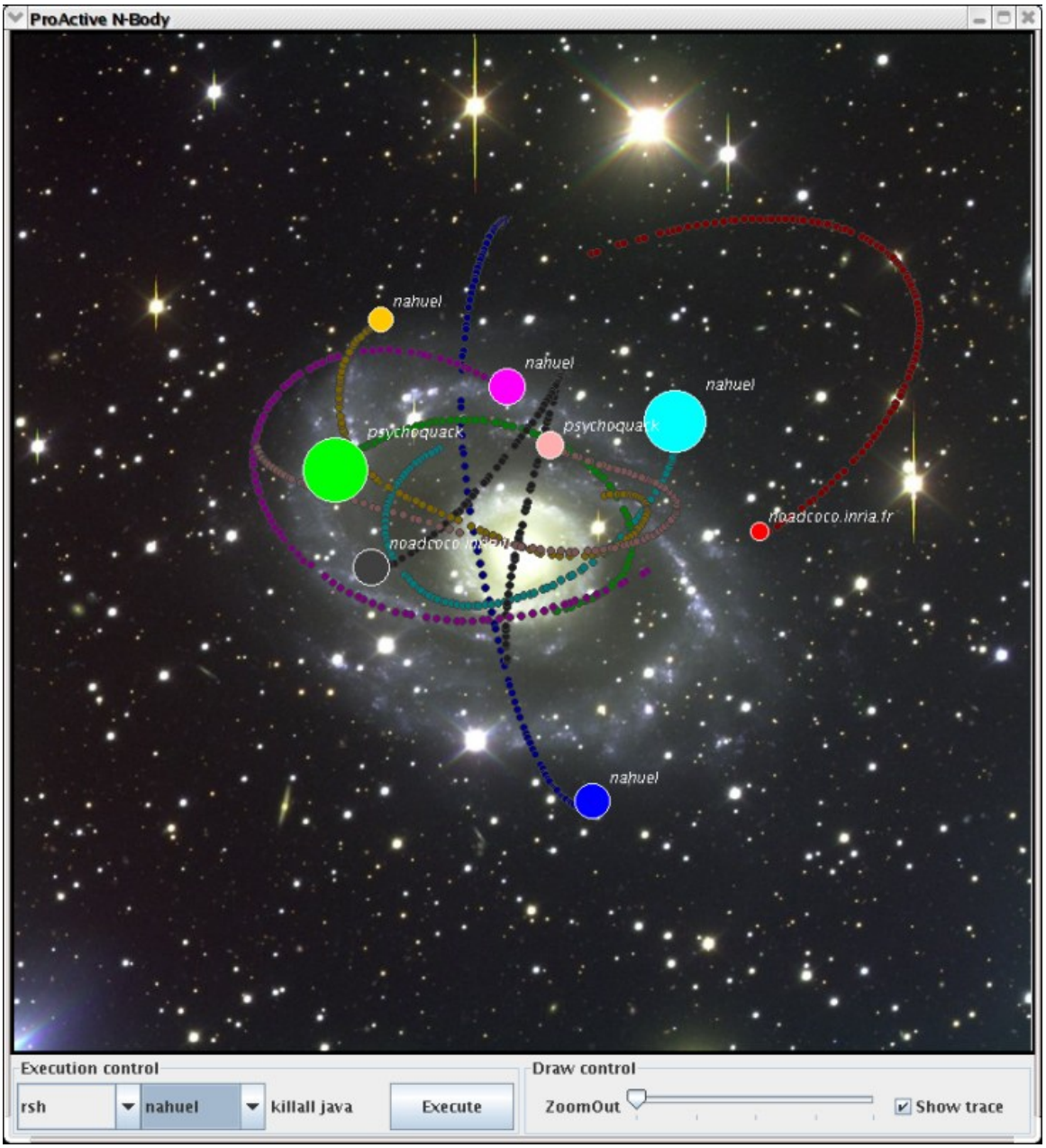
Fault-tolerance in ProActive is still not compliant with the following features :

- active objects exposed as Web services [WSDoc.xml], or reachable using http protocol,
- and security [./Security.xml], as fault-tolerance servers are implemented using standard RMI.

13.4. A complete example

13.4.1. Description

You can find in `ProActive/scripts/[unix|windows]/ft/nbodyft.[sh|bat]` a script that starts a fault-tolerant version of the ProActive NBody [http://www-sop.inria.fr/oasis/ProActive/apps/nbody.xml] example. This script actually call the `ProActive/scripts/[unix|windows]/nbody.[sh|bat]` script with the option `-displayft`. The java source code is the same as the standard version. The only difference is the "Execution Control" panel added in the graphical interface, which allows the user to remotely kill Java Virtual Machine so as to trigger a failure by sending a `killall java` signal. Note that this panel will not work with Windows operating system, since the `killall` does not exist. But a failure can be triggered for example by killing the JVM process on one of the hosts.



This snapshot shows a fault-tolerant execution with 8 bodies on 3 different hosts. Clicking on the "Execute" button will trigger the failure of the host called Nahuel and the recovery of the 8 bodies. The checkbox *Show trace* is checked: the 100 latest positions of each body are drawn with darker points. These traces allow to verify that, after a failure, each body finally reach the position it had just before the failure.

13.4.2. Running NBody example

Before starting the fault-tolerant body example, you have to edit the `ProActive/descriptors/FaultTolerantWorkers.` deployment descriptor so as to deploy on your own hosts (**HOSTNAME**), as follow:

```
...
 <processDefinition id="jvmAppli1">
   <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHJVMProcess"
hostname="HOSTNAME">
    <processReference refid="jvmProcess"/>
  </rshProcess>
 </processDefinition>
...
```

Of course, more than one host is needed to run this example, as failure are triggered by killing all Java processes on the selected host.

The deployment descriptor must also specify the GlobalFTServer location as follow, assuming that the script `startGlobalFTServer.sh` has been started on the host **SERVER_HOSTAME**:

```
...
 <services>
  <serviceDefinition id="appli">
   <faultTolerance>
   <protocol type="cic"></protocol>
    <globalServer
url="rmi://SERVER_HOSTAME:1100/FTServer"></globalServer>
    <ttc value="5"></ttc>
   </faultTolerance>
  </serviceDefinition>
  <serviceDefinition id="ressource">
   <faultTolerance>
   <protocol type="cic"></protocol>
    <globalServer
url="rmi://SERVER_HOSTAME:1100/FTServer"></globalServer>
    <resourceServer
url="rmi://SERVER_HOSTAME:1100/FTServer"></resourceServer>
    <ttc value="5"></ttc>
   </faultTolerance>
  </serviceDefinition>
 </services>
...
```

Finally, you can start the fault-tolerant ProActive NBody and choose the version you want to run :

```
~/ProActive/scripts/unix/FT> ./nbodyFT.sh
```

```
Starting Fault-Tolerant version of ProActive NBody...
--- N-body with ProActive --------------------------------
 **WARNING** : $PROACTIVE/descriptors/FaultTolerantWorkers.xml MUST BE SET \
WITH EXISTING HOSTNAMES !
      Running with options set to 4 bodies, 3000 iterations, display true
 1 : Simplest version, one-to-one communication and master
 2 : group communication and master
 3 : group communication, odd-even-synchronization
 4 : group communication, oospmd synchronization
 5 : Barnes-Hut, and oospmd
Choose which version you want to run [12345] :
4
Thank you!
 --> This ClassFileServer is reading resources from classpath
Jini enabled
Ibis enabled
Created a new registry on port 1099
//tranquility.inria.fr/Node-157559959 successfully bound in registry at //t\
ranquility.inria.fr/Node-157559959
Generating class : pa.stub.org.objectweb.proactive.examples.nbody.common.St\
ub_Displayer
************* Reading deployment descriptor: file:./../../../descriptors/\
FaultTolerantWorkers.xml ********************
```

Part IV. Composing

Table of Contents

Chapter 14. Components introduction

Computing Grids and Peer-to-Peer networks are inherently heterogeneous and distributed, and for this reason they drive new technological challenges : complexity in the design of applications, complexity of deployment, reusability and performance issues. The objective of our work is to provide an answer to these problems through the implementation for ProActive of an extensible, dynamical and hierarchical component model, Fractal [http://fractal.objectweb.org]. This document is an overview of the implementation of Fractal with ProActive. First, it explains the goals and the reasons for a new implementation of the Fractal model. Second, it shows the extensions to the model and the conformance to the Fractal specification. The third section is a word on the current Architecture Description Language. The fourth section goes through some examples to illustrate the use of the API and the distribution of components. The final section is about our forthcoming research work. You can also get an overview of the architecture of this implementation here [architecture.xml].

For a general overview of our work , you can also refer to a paper presented at the International Symposium on Distributed Objects and Applications (DOA), in November 2003. (.pdf [http://www-sop.inria.fr/oasis/ProActive/doc/HierarchicalGridComponents.pdf])

Chapter 15. An implementation of the Fractal component model with ProActive

Fractal defines a general conceptual model, along with a programming application interface (API) in Java. According to the official documentation, the Fractal component model is "*a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces*".

There is a reference implementation, called Julia.

We first tried to use Julia to manipulate active objects (the fundamental entities in ProActive), but we wouldn't have been able to reuse the features of the Proactive library, because of the architectures of the libraries.

Julia manipulates a base class by modifying the bytecode or adding interception objects to it. On the other hand, ProActive is based on a meta-object protocol and provides a reference to an active object through a typed stub. If we wanted to use active objects with Julia, the Julia runtime would try to manipulate the stub, and not the active object itself. And if trying to force Julia to work on the same base object than ProActive, the control flow could not traverse both ProActive and Julia.

Eventually, re-implementing ProActive using Julia could be a solution (a starting point could be the "protoactive" example of Julia), but this would imply a full refactoring of the library, and therefore quite a few resources...

More generally speaking, Julia is designed to work with standard objects, but not with the active objects of ProActive. Some features (see next section) would not be reusable using Julia with ProActive active objects.

Therefore, we had to go for our own implementation.

This implementation is different from Julia both in its objectives and in the programming techniques. As previously stated, we target Grid and P2P environments. The programming techniques and the architecture of the implementation is described in a following section.

15.1. Specific features of this implementation

The combination of the Fractal model with the ProActive library leverages the capabilities of both of them.

15.1.1. Distribution

Distribution is achieved in a transparent manner over the Java RMI protocol thanks to the use of a stub/proxy pattern. Components are manipulated indifferently of their location (local or on a remote JVM).

15.1.2. Deployment framework

ProActive provides a deployment framework for creating a distributed component system. Using a configuration file and the concept of virtual nodes, this framework :

1. connects to remote hosts using supported protocols, such as rsh, rlogin, ssh, globus, lsf etc...
2. creates JVMs on these hosts
3. instantiates components on these newly created JVMs

15.1.3. Activities

A fundamental concept of the ProActive library is this of active objects [../ProActiveBasis.xml], where activities can actually be redefined [../ActiveObjectCreation.xml] so as to customize their **behavior**.

15.1.4. Asynchronous method calls with futures

Asynchronous method calls with transparent futures [../FutureObjectCreation.xmlAsynchronous] is a core feature of ProActive, and it allows concurrent processing. Indeed, suppose a caller invokes a method on a callee. This method returns a result on a component. With synchronous method calls, the flow of execution of the caller is blocked until the result of the method called is received. In the case of intensive computations, this can be relatively long. With asynchronous method calls, the caller gets a future object and will continue its tasks until it really uses the result of the method call. The process is then blocked (it is called wait-by-necessity) until the result has effectively been calculated.

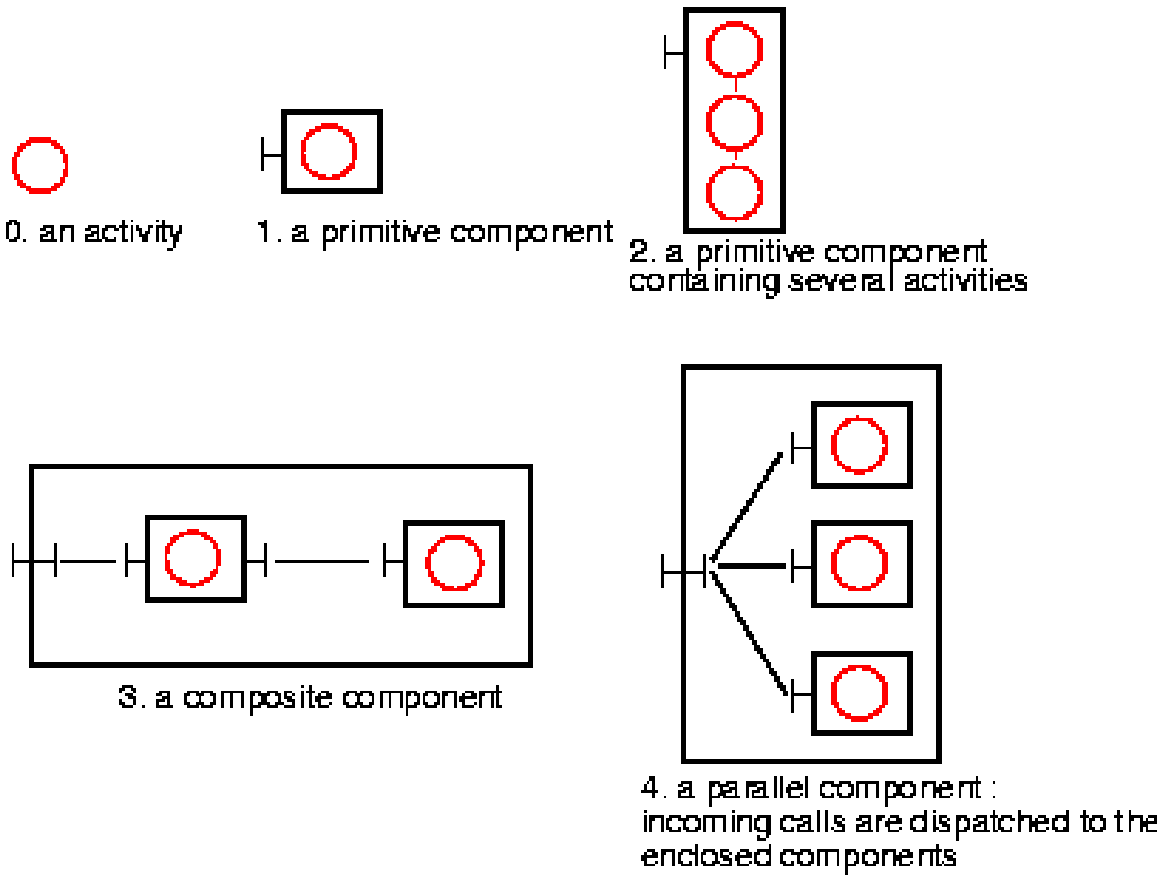
Chapter 16. Conformance to the Fractal model and extensions

16.1. Model

The conceptual model of Fractal is the base of our work, and as it is extensible, we have added a few concepts to fulfill our needs. The Fractal specification defines conformance levels for implementations of the API (section 7.1. of the Fractal 2 specification). The implementation for ProActive is conformant up to level 3.2. . In other words, it is fully compliant with the API, except it does not consider the creation of components through template components.

The implementation for ProActive currently defines 2 extensions to the base component model :

- **Distributed deployment** : components can be deployed onto distributed virtual machines, using the deployment facilities of ProActive.
- **Parallel components** : this type of components is a specialization of the composite components. They encapsulate other components of the same type, and all incoming calls are forwarded to the corresponding internal interfaces of the enclosed components. This allows parallel processing while just manipulating one entity, the enclosing parallel component. As we use the typed groups API of ProActive, coupled with the concept of internal collective interfaces of Fractal, the communications to the enclosed components are either *scattered* or *broadcasted*. The following figure sums up the different kinds of components available.



The different kinds of components with the ProActive implementation. Primitive components have customizable activities (primitive components are also active objects).

16.2. Implementation specific API

16.2.1. fractal.provider

The API is the same for any Fractal implementation, though some classes are implementation-specific :

The fractal provider class, that corresponds to the `fractal.provider` parameters of the JVM, is `org.objectweb.proactive.core.component.Fractive`. The `Fractive` class acts as :

- a bootstrap component
- a GenericFactory for instantiating new components
- a utility class providing static methods to create collective interfaces and retreive references to ComponentParametersController

16.2.2. Content and controller descriptions

The controller description and the content description of the components, as specified in the method `public Component newFcInstance(Type type, Object controllerDesc, Object contentDesc) throws InstantiationException` of the `org.objectweb.fractal.api.factory.Factory` class, correspond in this implementation to the classes `org.objectweb.proactive.core.component.ControllerDes` and `org.proactive.core.component.ContentDescription`.

16.2.3. Collective bindings

In composite or parallel components, collective bindings are performed automatically. For primitive component, the developer has to implement the bindings explicitely in the code. We provide a method in the `org.objectweb.proactive.core.component.Fractal` class for creating collective bindings :

```
public static ProActiveInterface createCollectiveClientInterface(String itf\
Name, String itfSignature) throws ProActiveRuntimeException
```

where `itfName` is the name of the interface, and `itfSignature` is the signature of the interface.

Suppose you have an attribute of the base class of the primitive component of type I, named i. The initialization of the binding would be :

```
i= (I) Fractal.createCollectiveClientInterface("i",I.class.getName());
```

Then the binding method (implementation of the BindingController interface) would be :

```
public void bindFc(String clientItfName, Object serverItf) {
 if (clientItfName.equals("i")) {
 ProActiveGroup.getGroup(i).add(serverItf);
 }
```

```
}
```

You will be able to see the collective interface as an object of type I, and therefore invoke methods defined by I. But you will also be able to see the collective interface as a collection [http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.xml], as a group can also be manipulated as a collection :

```
Collection c = ProActiveGroup.getGroup(i);
```

16.2.4. Requirements

As this implementation is based on ProActive, several conditions [../ActiveObjectCreation.xml] are required :

- the base class for the implementation of a primitive component has to provide an empty, no-args constructor.
- parameters types and return types of the methods provided by the interfaces of the components have to be reifiable.

Chapter 17. Configuration

17.1. Controllers and interceptors

This section explains how to customize the membranes of component through the configuration, composition and creation of controllers and interceptors.

17.1.1. Configuration of controllers

It is possible to customize controllers, by specifying a control interface and an implementation.

Controllers are configured in a simple XML configuration file, which has the following structure :

```
<componentConfiguration>
 <controllers>
  <controller>
 <interface>ControllerInterface</interface>
    <implementation>ControllerImplementation</implementation>
  </controller>
...
```

Unless they some controllers are also interceptors (see later on), the controllers do not have to be ordered.

A default configuration file is provided, it defines the default controllers available for every ProActive component (super, binding, content, naming, lifecycle and component parameters controllers).

A custom configuration file can be specified (in this example with "thePathToMyConfigFile") for any component in the controller description parameter of the newFcInstance method from the Fractal API :

```
componentInstance = componentFactory.newFcInstance(
  myComponentType,
  new ControllerDescription(
    "name",
    myHierarchicalType,
    thePathToMyControllerConfigFile),
  myContentDescription);
```

17.1.2. Writing a custom controller

The controller interface is a standard interface which defines which methods are available.

When a new implementation is defined for a given controller interface, it has to conform to the following rules :

1. The controller implementation must extend the AbstractProActiveController class, which is the base class for component controllers in ProActive, and which defines the constructor AbstractProActiveController(Component owner).
2. The controller implementation must override this constructor :

```
public ControllerImplementation(Component owner) {
 super(owner);
}
```

1. The controller implementation must also override the abstract method setControllerItfType(), which sets the type of the controller interface :

```
protected void setControllerItfType() {
 try {
  setItfType(ProActiveTypeFactory.instance().createFcItfType(
    "Name of the controller",
    TypeFactory.SINGLE));
  } catch (InstantiationException e) {
    throw new ProActiveRuntimeException("cannot create controller type : " +
      this.getClass().getName());
  }
 }
```

1. The controller interface and its implementation have to be declared in the component configuration file.

17.1.3. Configuration of interceptors

Controllers can also act as interceptors : they can intercept incoming invocations and outgoing invocations. For each invocation, pre and post processings are defined in the methods beforeInputMethodInvocation, afterInputMethodInvocation, beforeOutputMethodInvocation, and afterOutputMethodInvocation. These methods are defined in the interfaces InputInterceptor and OutputInterceptor, and take a MethodCall object as an argument. MethodCall objects are reified representations of method invocations, and they contain Method objects, along with the parameters of the invocation.

Interceptors are configured in the controllers XML configuration file, by simply adding input-interceptor="true" or/and output-interceptor="true" as attributes of the controller element in the definition of a controller (provided of course the specified interceptor is an input or/and output interceptor). For example a controller that would be an input interceptor and an output interceptor would be defined as follows :

```
<componentConfiguration>
 <controllers>
 ....
  <controller
input-interceptor="true" output-interceptor="true"
>
    <interface>InterceptorControllerInterface</interface>
    <implementation>ControllerImplementation</implementation>
  </controller>
...
```

Interceptors can be composed in a basic manner : sequentially.

For input interceptors, the beforeInputMethodInvocation method is called sequentially for each controller in the order they are defined in the controllers configuration file. The afterInputMethodInvocation method is called sequentially for each controller in the reverse order they are defined in the controllers configuration file.

If in the controller config file, the list of input interceptors is in this order (the order in the controller config file is from top to bottom) :

```
InputInterceptor1
InputInterceptor2
```

This means that an invocation on a server interface will follow this path :

```
--> caller
--> InputInterceptor1.beforeInputMethodInvocation
--> InputInterceptor2.beforeInputMethodInvocation
--> callee.invocation
--> InputInterceptor2.afterInputMethodInvocation
--> InputInterceptor1.afterInputMethodInvocation
```

For output interceptors, the beforeOutputMethodInvocation method is called sequentially for each controller in the order they are defined in the controllers configuration file. The afterOutputMethodInvocationmethod is called sequentially for each controller in the reverse order they are defined in the

controllers configuration file.

If in the controller config file, the list of input interceptors is in this order (the order in the controller config file is from top to bottom) :

```
OutputInterceptor1
OutputInterceptor2
```

This means that an invocation on a server interface will follow this path

```
--> currentComponent
--> OutputInterceptor1.beforeOutputMethodInvocation
--> OutputInterceptor2.beforeOutputMethodInvocation
--> callee.invocation
--> OutputInterceptor2.afterOutputMethodInvocation
--> OutputInterceptor1.afterOutputMethodInvocation
```

17.1.4. Writing a custom interceptor

An interceptor being a controller, it must follow the rules explained above for the creation of a custom controller.

Input interceptors and output interceptors must implement respectively the interfaces InputInterceptor and OutputInterceptor, which declare interception methods (pre/post interception) that have to be implemented.

Here is a simple example of an input interceptor :

```
public class MyInputInterceptor extends AbstractProActiveController
 implements InputInterceptor, MyController {
  public MyInputInterceptor(Component owner) {
    super(owner);
  }

  protected void setControllerItfType() {
    try {
      setItfType(ProActiveTypeFactory.instance().createFcItfType("my control\
ler",
       MyController.class.getName(), TypeFactory.SERVER,
       TypeFactory.MANDATORY, TypeFactory.SINGLE));
    } catch (InstantiationException e) {
      throw new ProActiveRuntimeException("cannot create controller " +
       this.getClass().getName());
    }
  }
  // foo is defined in the MyController interface
  public void foo() {
    // foo implementation
  }
  public void afterInputMethodInvocation(MethodCall methodCall) {
    System.out.println("post processing an intercepted an incoming functiona\
l invocation");
    // interception code
  }
  public void beforeInputMethodInvocation(MethodCall methodCall) {
    System.out.println("pre processing an intercepted an incoming functional\
 invocation");
    // interception code
 }
}
```

The configuration file would state :

```
<componentConfiguration>
 <controllers>
 ....
   <controller
input-interceptor="true"
>
    <interface>
MyController
</interface>
    <implementation>
MyInputInterceptor
</implementation>
   </controller>
   ...
```

17.2. Lifecycle : encapsulation of functional activity in component lifecycle

In this implementation of the Fractal component model, Fractal components are active objects. Therefore it is possible to redefine their activity. In this context of component based programming, we call an activity redefined by a user a functional activity.

When a component is instantiated, its lifecycle is in the STOPPED state, and the functional activity that a user may have redefined is not started yet. Internally, there is a default activity which handles controller requests in a FIFO order.

When the component is started, its lifecycle goes to the STARTED state, and then the functional activity is started : this activity is initialized (as defined in InitActive), and run (as defined in RunActive).

2 conditions are required for a smooth integration between custom management of functional activities and lifecycle of the component :

1. the control of the request queue must use the org.objectweb.proactive.Service class
2. the functional activity must loop on the body.isActive() condition (this is not compulsory, but it allows to automatically end the functional activity when the lifecycle of the component is stopped. It may also be managed with a custom filter).

Control invocations to stop the component will automatically set the isActive() return value to false, which implies that when the functional activity loops on the body.isActive() condition, it will end when the lifecycle of the component is set to STOPPED.

17.3. Shortcuts

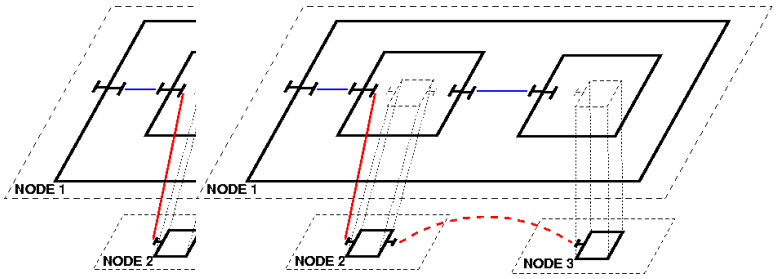
17.3.1. Principles

Communications between components in a hierarchical model may involve the crossing of several membranes, and therefore paying the cost of several indirections. If the invocations are not intercepted in the membranes, then it is possible to optimize the communication path by shortcutting : communicating directly from a caller component to a callee component by avoiding indirections in the membranes.

In the Julia implementation, a shortcut mechanism is provided for components in the same JVM, and the implementation of this mechanism relies on code generation techniques.

We provide a shortcut mechanism for distributed components, and the implementation of this mechanism relies on a "tensioning" technique : the first invocation determines the shortcut path, then the following invocations will use this shortcut path.

For example, in the following pictures, the first invocation from the only client interface of component in node2 follows the standard communication path, as defined during the binding process, and the invocation eventually reaches the server interface of the component in node 3. If no interception takes place during the transfer of the invocation, then a shortcut is possible, and further invocations from the client interface of component in node2 will go directly to the server interface of the component in node 3.



a configuration without shorctut

a configuration with shorctut between the components in node2 and node3

17.3.2. Configuration

Shortcuts are available when composite components are synchronous components (this does not break the ProActive model, as composite components are structural components). Components can be specified as synchronous in the ControllerDescription object that is passed to the component factory :

```
ControllerDescription controllerDescription =
          new ControllerDescription("name", Constants.COMPOSITE,↩
Constants.SYNCHRONOUS);
```

When the system property proactive.components.use_shortcuts is set to true, the component system will automatically establish shortcuts between components whenever possible.

Chapter 18. Architecture Description Language

The Architecture Description Language (ADL) is used to configure and deploy component systems. The architecture of the system is described in a normalized XML file.

The ADL has been updated and is now an extension of the standard Fractal ADL, allowing to reuse ProActive-specific features such as distributed deployment using deployment descriptors.

The distributed deployment facilities offered by ProActive are reused, and the notion of virtual node is integrated in the component ADL. For this reason, the components ADL has to be associated with a deployment descriptor (this is done at parsing time : both files are given to the parser).

One should refer to the Fractal ADL tutorial [http://fractal.objectweb.org/tutorials/adl/index.xml] for more detailed information about the ADL. Here is a short overview, and a presentation of some added features.

Note that because this ADL is based on the Fractal ADL, it requires the following libraries (included in the /lib directory of the ProActive distribution) : fractal-adl.jar, dtdparser.jar, ow_deployment_scheduling.jar

18.1. Overview

Components are defined in **definition** files, which are .fractal files. The syntax of the document is validated against a DTD retreived from the classpath

```
classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd
```

The **definition** element has a name (which must be the same name that the file's) and inheritance is supported through the attribute "extends" :

```
definition
name="org.objectweb.proactive.examples.components.helloworld.hell\
oworld-distributed-wrappers"
```

The exportedVirtualNodes elements is described later in this section

Components can be specified and created in this definition, and these components can themselves be defined in other definition files :

```
component name="client-wrapper"
definition="org.objectweb.proactive.examples.c\
omponents.helloworld.ClientType"
```

Nesting is allowed for composite components and is done by adding other "component" elements.

The **binding** element specifies bindings between interfaces of components ", and specifying "this" as the name of the component refers to the current enclosing component.

```
binding client="this.r" server="client.r"/
```

The **controller** elements can have the following "desc" values : "composite", "parallel" or "primitive". A parallel component and the components it contains should be type-compatible

Primitive components specify the **content** element, which indicates the implementation class containing the business logic for this component :

```
content
class="org.objectweb.proactive.examples.components.helloworld.Client\
Impl"
```

The **virtual-node** element offers distributed deployment information. It can be exported and composed in the exportedVirtualNodes element.

The component will be instantiated on the virtual node it specified (or the one that it exported). For a composite or a parallel component, it means it will be instantiated on the (first if there are several nodes mapped) node of the virtual node. For a primitive component, if the virtual node defines several nodes (cardinality="multiple"), there will be as many instances of the primitive component as there are underlying nodes. Each of these instances will have a suffixed name looking like : "primiveComponentName-cyclicInstanceNumber-n", where primitiveComponentName is the name defined in the ADL. This automatic replication is used in the parallel components.

```
virtual-node name="client-node" cardinality="single"
```

The syntax is similar to the standard Fractal ADL, and the parsing engine has been extended. Features specific to ProActive are :

- Virtual nodes have a cardinality property : either "single" or "multiple". When "single", it means the virtual node in the deployment descriptor should contain 1 node ; when "multiple", it means the virtual node in the deployment descriptor should contain more than 1 node.
- Virtual nodes can be **exported** and **composed**.
- Template components are not handled.
- The controller description includes "parallel" as a valid attribute.
- The validating DTD has to be specified as : classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd

18.2. Example

The easiest way to understand the ADL is to see an example [helloworld-distributed.xml]. It corresponds to the helloworld example described later in this document.

18.3. Exportation and composition of virtual nodes

Components are deployed on the virtual node that is specified in their definition ; it has to appear in the deployment descriptor unless this virtual node is exported. In this case, the name of the exported virtual node should appear in the deployment descriptor, unless this exported virtual node is itself exported.

When exported, a virtual node can take part in the composition of other exported virtual nodes. The idea is to further extend reusability of existing (and packaged, packaging being a forthcoming feature of Fractal) components.

In the example, the component defined in helloworld-distributed-wrappers.fractal exports the virtual nodes VN1 and VN2:

```
exportedVirtualNodes
 exportedVirtualNode name="VN1"
  composedFrom
   composingVirtualNode component="client" name="client-node"
  /composedFrom
 /exportedVirtualNode
 exportedVirtualNode name="VN2"
  composedFrom
   composingVirtualNode component="server" name="server-node"/
  /composedFrom
 /exportedVirtualNode
/exportedVirtualNodes
```

VN1 is composed of the exported virtual node "client-node" from the component named client

In the definition of the client component (ClientImpl.fractal), we can see that client-node is an exportation of a virtual node which is also name "client-node" :

```
exportedVirtualNodes
 exportedVirtualNode name="client-node"
   composedFrom
     composingVirtualNode component="this" name="client-node"/
   /composedFrom
 /exportedVirtualNode
/exportedVirtualNodes
...
virtual-node name="client-node" cardinality="single"/
```

Although this is a simplistic example, one should foresee a situation where ClientImpl would be a prepackaged component, where its ADL could not be modified ; the exportation and composition of virtual nodes allow to adapt the deployment of the system depending on the existing infrastructure. Colocation can be specified in the enclosing component definition (helloworld-distributed-wrappers.fractal) :

```
exportedVirtualNodes
 exportedVirtualNode name="VN1"
  composedFrom
   composingVirtualNode component="client" name="client-node"
   composingVirtualNode component="server" name="server-node"/
  /composedFrom
 /exportedVirtualNode
/exportedVirtualNodes
```

As a result, the client and server component will be colocated / deployed on the same virtual node. This can be profitable if there is a lot of communications between these two components.

When specifying "null" as the name of an exported virtual node, the components will be deployed on the current virtual machine. This can be useful for debugging purposes.

18.4. Usage

The launcher, which parses the ADL, creates a corresponding component factory, and instantiates and assembles the components as defined in the ADL, is started from the `org.objectweb.proactive.core.component.adl.Launche` class :

```
Launcher [-java|-fractal] <definition> [ <itf> ] [deployment-descriptor])
```

where [-java|-fractal] comes from the Fractal ADL Launcher (put -fractal for ProActive components, this will be made optional for ProActive components in the next release), <definition> is the name of the component to be instantiated and started, <itf> is the name of its Runnable interface, if it has one, and <deployment-descriptor> the location of the ProActive deployment descriptor to use. It is also possible to use this class directly from its static main method.

Chapter 19. Examples

3 examples are presented : code snippets for visualizing the transition between active objects and components, the "hello world", from the Fractal tutorial, and C3D component version. The programming model is Fractal, and one should refer to the Fractal documentation for detailed examples.

19.1. From objects to active objects to distributed components

In Java, objects are created by instantiation of classes. With ProActive, one can create active objects from Java classes, while components are created from component definitions. Let us first consider the "A" interface :

```
public interface A {
  public String foo(); // dummy method
}
```

"AImpl" is the class implementing this interface :

```
public class AImpl implements A {
 public AImpl() {}
 public String foo() {
 // do something
 }
}
```

The class is then instantiated in a standard way :

```
A object = new AImpl();
```

Active objects are instantiated using factory methods from the ProActive class (see the ProActive Hello World example [../HelloWorld.xml]). It is also possible to specify the activity of the active object, the location (node or virtual node), or a factory for meta-objects, using the appropriate factory method.

```
A active_object = (A)ProActive.newActive(
 AImpl, // signature of the base class
 new Object[] {}, // Object[]
 aNode, // location, could also be a virtual node
);
```

As components are also active objects in this implementation, they benefit from the same features, and are configurable in a similar way. Constructor parameters, nodes, activity, or factories, that can be specified for active objects, are also specifiable for components. The definition of a component requires 3 sub-definitions : the type, the description of the content, and the description of the controller.

19.1.1. Type

The type of the component (i.e. the functional interfaces provided and required) is specified in a standard way : (as taken from the Fractal tutorial)

We begin by creating objects that represent the types of the components of the application. In order to do this, we must first get a bootstrap component. The standard way to do this is the following one (this method creates an instance of the class specified in the fractal.provider system property, and uses this instance to get the bootstrap component):

```
Component boot = Fractal.getBootstrapComponent();
```

We then get the TypeFactory interface provided by this bootstrap component:

```
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
```

We can then create the type of the first component, which only provides a A server interface named "a":

```
// type of the a component
ComponentType aType = tf.createFcType(new InterfaceType[] {
 tf.createFcItfType("a", "A", false, false, false)
});
```

19.1.2. Description of the content

The second step in the definition of a component is the definition of its content. In this implementation, this is done through the ContentDescription class :

```
ContentDescription contentDesc = new ContentDescription(
 AImpl, // signature of the base class
 new Object[] {}, // Object[]
 aNode, // location, could also be a virtual node
);
```

19.1.3. Description of the controller

Properties relative to the controller can be specified in the ControllerDescription :

```
ControllerDescription controllerDesc = new ControllerDescription(
 "myName", // name of the component
 Constants.PRIMITIVE // the hierarchical type of the component
 // it could be PRIMITIVE, COMPOSITE, or PARALLEL
);
```

Eventually, the component definition is instantiated using the standard Fractal API. This component can then be manipulated as any other Fractal component.

```
Component component = componentFactory.newFcInstance(
 componentType, // type of the component (defining the client and server in\
terfaces)
 controllerDesc, // implementation-specific description for the controller
 contentDesc // implementation-specific description for the content
);
```

19.1.4. From attributes to client interfaces

There are 2 kinds of interfaces for a component : those that offer services, and those that require services. They are named respectively server and client interfaces.

From a Java class, it is fairly natural to identify server interfaces : they (can) correspond to the Java interfaces implemented by the class. In the above example, "a" is the name of an interface provided by the component, corresponding to the "A" Java interface.

On the other hand, client interfaces usually correspond to attributes of the class, in the case of a primitive component. If the component defined above requires a service from another component, say the one corresponding to the "Service" Java interface, the AImpl class should be modified. As we use the *inversion of control* pattern, a BindingController is provided, and a binding operation on the "requiredService" interface will actually set the value of the "service" attribute, of type "Service".

First, the type of the component is changed :

```
// type of the a component
ComponentType aType = tf.createFcType(new InterfaceType[] {
 tf.createFcItfType("a", "A", false, false, false),
 tf.createFcItfType("requiredService", "A", true, false, false)
});
```

The Service interface is the following :

```
// The Service interface
public interface Service {
 public String bar();
}
```

And the AImpl class is :

```
// The modified AImpl class
public class AImpl implements A, BindingController {
 Service service; // attribute corresponding to a client interface
 public AImpl() {}
 // implementation of the A interface
 public String foo() {
   return s.bar(); // for example
 }
 // implementation of BindingController
 public Object lookupFc (final String cItf) {
   if (cItf.equals("requiredService")) {
```

```
    return service;
  }
  return null;
 }
 // implementation of BindingController
 public void bindFc (final String cItf, final Object sItf) {
  if (cItf.equals("requiredService")) {
    service = (Service)sItf;
  }
 }
 // implementation of BindingController
 public void unbindFc (final String cItf) {
  if (cItf.equals("requiredService")) {
    service = null;
  }
 }
}
```

19.2. The HelloWorld example

The mandatory helloworld example (from the Fractal tutorial) shows the different ways of creating a component system (programmatically and using the ADL), and it can easily be implemented using ProActive.

19.2.1. Set-up

You can find the code for this example in the package org.objectweb.proactive.examples.components.helloworld of the ProActive distribution.

The code is almost identical to the Fractal tutorial's example [http://fractal.objectweb.org/tutorials/fractal/index.xml].

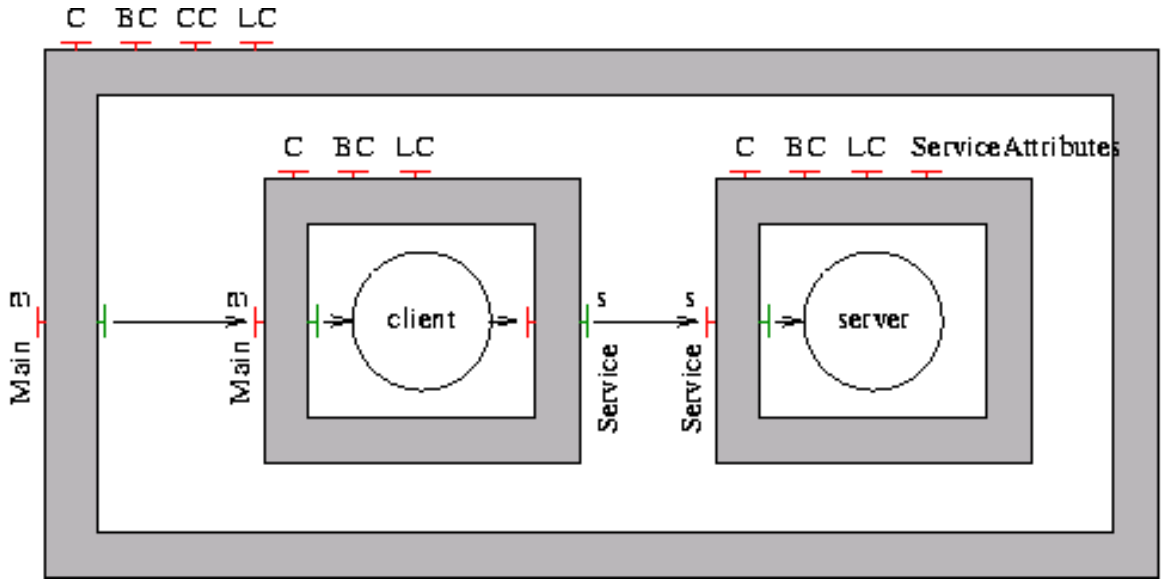
The differences are the following :

- The reference example is provided for level 3.3. implementation, whereas this current implementation is compliant up to level 3.2 : templates are not provided. Thus you will have to skip the specific code for templates.
- The `newFcInstance` method of the `GenericFactory` interface, used for directly creating components, takes 2 implementation-specific parameters. So you should use the `org.objectweb.proactive.component.ControllerDescription` and `org.objectweb.proactive.com` classes to define ProActive components. (It is possible to use the same parameters than in Julia, but that hinders you from using some functionalities specific to ProActive, such as distributed deployment or definition of the activity).
- Collective interfaces could be implemented the same way than suggested, but using the `Fractive.createCollectiveClientInterface` method will prove useful with this implementation : you are then able to use the functionalities provided by the typed groups API.
- Components can be distributed
- the ClientImpl provides an empty no-args constructor.

19.2.2. Architecture

The helloworld example is a simple client-server application, where the client (c) and the server (s) are components, and they are both contained in the same root component (root).

Another configuration is also possible, where client and server are wrapped around composite components (C and S). The goal was initially to show the interception shortcut mechanism in Julia. In the current ProActive implementation, there are no such shortcuts, as the different components can be distributed, and all invocations are intercepted. The exercise is still of interest, as it involves composite components.



19.2.3. Distributed deployment

This section is specific to the ProActive implementation, as it uses the deployment framework of this library.

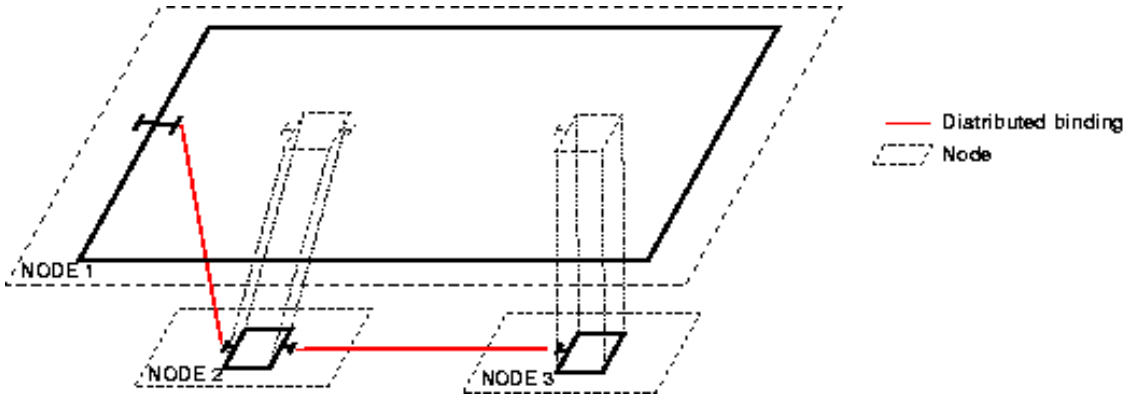
If the application is started with (only) the parameter "distributed", the ADL used is "helloworld-distributed.fractal", where virtualNode of the client and server components are exported as VN1 and VN2. Exported virtual node names from the ADL match those defined in the deployment descriptor "deployment.xml".

One can of course customize the deployment descriptor and deploy components onto virtually any computer, provided it is connectable by supported protocols. Supported protocols include LAN, clusters and Grid protocols (see deployment descriptors documentation [../Descriptor.xml]).

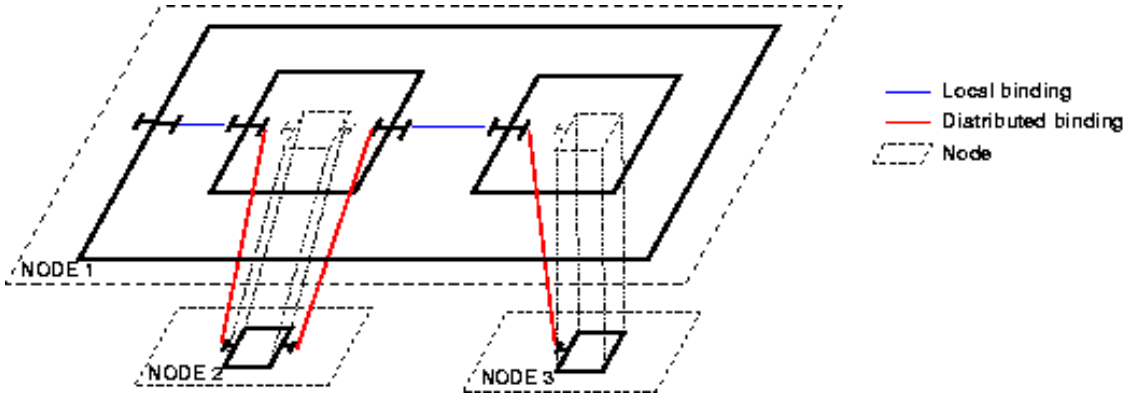
Have a look at the ADL files "helloworld-distributed.fractal" and "helloworld-distributed-wrappers.fractal". In a nutshell, they say : "the primitive components of the application (client and server) will run on given exported virtual nodes, whereas the other components (wrappers, root component) will run on the current JVM.

Therefore, we have the two following configurations :

1. The one without wrappers, where the primitive components are distributed.



2. The one with wrappers, where again, only the primitive components are distributed.



Currently, bindings are not optimized. For example, in the configuration with wrappers, there is an indirection that can be costly, between the client and the server. We are currently working on optimizations that would allow to shortcut communications, while still allowing coherent dynamic reconfiguration. It is the same idea than in Julia, but we are dealing here with distributed components. It could imply compromises between dynamicity and performance issues.

19.2.4. Execution

You can either compile and run the code yourself, or follow the instructions for preparing the examples and use the script helloworld_fractal.sh (or .bat). If you choose the first solution, do not forget to set the fractal.provider system property.

If you run the program with no arguments (i.e. not using the parser, no wrapper composite components, and local deployment) , you should get something like this:

```
01 --> This ClassFileServer is reading resources from classpath
02 Jini enabled
03 Ibis enabled
04 Created a new registry on port 1099
05 //crusoe.inria.fr/Node363257273 successfully bound in registry at //crus\
oe.inria.fr/Node363257273
06 Generating class : pa.stub.org.objectweb.proactive.core.component.type.S\
tub_Composite
```

```
07 Generating class : pa.stub.org.objectweb.proactive.examples.components.h\
elloworld.Stub_ClientImpl
08 Generating class : pa.stub.org.objectweb.proactive.examples.components.h\
elloworld.Stub_ServerImpl
```

You can see :

- line 01 : the creation of the class file server which handles the on-the-fly generation and distribution of ProActive stubs and component functional interfaces
- line 04 : the creation of a rmi registry
- line 05 : the registration of the default runtime node
- line 06 to 08 : the on-the-fly generation of ProActive stubs (the generation of component functional interfaces is silent)

Then you have (the exception that pops out is actually the expected result, and is intended to show the execution path) :

```
01 Server: print method called
02 at org.objectweb.proactive.examples.components.helloworld.ServerImpl.pri\
nt(ServerImpl.java:37)
03 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
04 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.\
java:39)
05 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces\
sorImpl.java:25)
06 at java.lang.reflect.Method.invoke(Method.java:324)
07 at org.objectweb.proactive.core.mop.MethodCall.execute(MethodCall.java:3\
73)
08 at org.objectweb.proactive.core.component.request.ComponentRequestImpl.s\
erveInternal(ComponentRequestImpl.java:163)
09 at org.objectweb.proactive.core.body.request.RequestImpl.serve(RequestIm\
pl.java:108)
10 at org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.se\
rve(BodyImpl.java:297)
11 at org.objectweb.proactive.core.body.AbstractBody.serve(AbstractBody.jav\
a:799)
12 at org.objectweb.proactive.core.body.ActiveBody$FIFORunActive.runActivit\
y(ActiveBody.java:230)
13 at org.objectweb.proactive.core.body.ActiveBody.run(ActiveBody.java:145)
14 at java.lang.Thread.run(Thread.java:534)
15 Server: begin printing...
16 --------> hello world
17 Server: print done.
```

What can be seen is very different from the output you would get with the Julia implementation. Here is what happens (from bottom to top of the stack):

- line 14 : The active object runs its activity in its own Thread
- line 12 : The default activity is to serve incoming request in a FIFO order

- line 08 : Requests (reified method calls) are encapsulated in ComponentRequestImpl objects
- line 06 : A request is served using reflection
- line 02 : The method invoked is the print method of an instance of ServerImpl

Now let us have a look at the distributed deployment : execute the program with the parameters "distributed parser". You should get something similar to the following :

```
01 --> This ClassFileServer is reading resources from classpath
02 Jini enabled
03 Ibis enabled
04 Created a new registry on port 1099
05 ************* Reading deployment descriptor: file:/0/user/mmorel/ProActi\
ve/classes/org/objectweb/proactive/examplescomponents/helloworld/deployment\

.xml ********************
06 created VirtualNode name=VN1
07 created VirtualNode name=VN2
08 created VirtualNode name=VN3
09 **** Starting jvm on crusoe.inria.fr
10 --> This ClassFileServer is reading resources from classpath
11 Jini enabled
12 Ibis enabled
13 Detected an existing RMI Registry on port 1099
14 //crusoe.inria.fr/VN1462549848 successfully bound in registry at //cruso\
e.inria.fr/VN1462549848
15 **** Mapping VirtualNode VN1 with Node: //crusoe.inria.fr/VN1462549848 d\
one
16 Generating class : pa.stub.org.objectweb.proactive.examples.components.h\
elloworld.Stub_ClientImpl
17 **** Starting jvm on crusoe.inria.fr
18 --> This ClassFileServer is reading resources from classpath
19 Jini enabled
20 Ibis enabled
21 Detected an existing RMI Registry on port 1099
22 //crusoe.inria.fr/VN21334775605 successfully bound in registry at //crus\
oe.inria.fr/VN21334775605
23 **** Mapping VirtualNode VN2 with Node: //crusoe.inria.fr/VN21334775605 \
done
24 Generating class : pa.stub.org.objectweb.proactive.examples.components.h\
elloworld.Stub_ServerImpl
25 //crusoe.inria.fr/Node1145479146 successfully bound in registry at //cru\
soe.inria.fr/Node1145479146
26 Generating class : pa.stub.org.objectweb.proactive.core.component.type.S\
tub_Composite
27 MOPClassLoader: class not found, trying to generate it
28 ClassServer sent class Generated_java_lang_Runnable_r_representative suc\
cessfully
39 MOPClassLoader: class not found, trying to generate it
30 ClassServer sent class Generated_java_lang_Runnable_r_representative suc\
cessfully
31 MOPClassLoader: class not found, trying to generate it
32 ClassServer sent class Generated_org_objectweb_proactive_examples_compon\
ents_helloworld_Service_s_representative successfully
33 MOPClassLoader: class not found, trying to generate it
34 ClassServer sent class Generated_org_objectweb_proactive_examples_compon\
```

```
ents_helloworld_ServiceAttributes_attribute_controller_representative↩
succe\
ssfully
35 ClassServer sent class pa.stub.org.objectweb.proactive.examples.componen\
ts.helloworld.Stub_ServerImpl successfully
```

What is new is :

- line 05 the parsing of the deployment descriptor
- line 09 and 17 : the creation of 2 virtual machines on the host "crusoe.inria.fr"
- line 15 and 24 : the mapping of virtual nodes VN1 and VN2 to the nodes specified in the deployment descriptor
- line 35 : the dynamic downloading of the stub class for ServerImpl: the stub class loader does not find the classes of the stubs in the current VM, and fetches the classes from the ClassServer
- line 28, 30, 32, 34 : the dynamic downloading of the classes corresponding to the components functional interfaces (they were silently generated)

Then we get the same output than for a local deployment, the activity of active objects is independent from its location.

```
01 Server: print method called
02 at org.objectweb.proactive.examples.components.helloworld.ServerImpl.pri\
nt(ServerImpl.java:37)
03 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
04 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.\
java:39)
05 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces\
sorImpl.java:25)
06 at java.lang.reflect.Method.invoke(Method.java:324)
07 at org.objectweb.proactive.core.mop.MethodCall.execute(MethodCall.java:3\
73)
08 at org.objectweb.proactive.core.component.request.ComponentRequestImpl.s\
erveInternal(ComponentRequestImpl.java:163)
09 at org.objectweb.proactive.core.body.request.RequestImpl.serve(RequestIm\
pl.java:108)
10 at org.objectweb.proactive.core.body.BodyImpl$ActiveLocalBodyStrategy.se\
rve(BodyImpl.java:297)
11 at org.objectweb.proactive.core.body.AbstractBody.serve(AbstractBody.jav\
a:799)
12 at org.objectweb.proactive.core.body.ActiveBody$FIFORunActive.runActivit\
y(ActiveBody.java:230)
13 at org.objectweb.proactive.core.body.ActiveBody.run(ActiveBody.java:145)
14 at java.lang.Thread.run(Thread.java:534)
15 Server: begin printing...
16 ->hello world
17 Server: print done.
```

19.3. The Comanche example

The Comanche example [http://fractal.objectweb.org/tutorial/index.xml] is a nice introduction to component based development with Fractal. It explains how to design applications using components, and how to implement these applications using the Fractal API.

You will notice that the example presented in this tutorial is based on Comanche, a simplistic http server. However, this example extensively uses reference passing through components. For example `Request` objects are passed by reference. This is incompatible with the ProActive programming model, where, to avoid shared passive objects, all passive objects passed to active objects are actually **passed by copy**(see ProActive basis [http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/ProActiveBasis.xml]). As active objects are themselves passed by reference, one could argue that we could turn some passive object into active objects. This would allow remote referencing through stubs. Unfortunately, for reasons specific to the Sockets and Streams implementations, (Socket streams implementations do not provide empty no-arg constructors), it is not easily possible to encapsulate some of the needed resource classes into active objects.

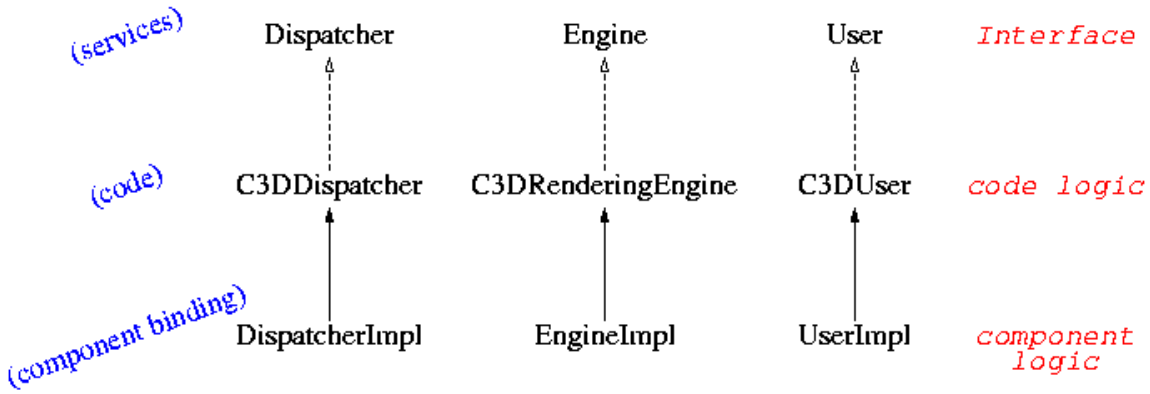
19.4. C3D - from Active Objects to Components

19.4.1. Reason for this example

This is an example of an application that is refactored to fit the components dogma. The standard C3D example has been taken as a basis, and component wrappers have been created. This way, one can see what is needed to transform an application into component-oriented code.

You may find the code in the examples/components/c3d [../doc/ProActive_src_html/org.objectweb.proactive.examples.components.c directory of the proactive source.

19.4.2. Using working C3D code with components



We consider the working C3D application. It's nice, and has a sleak GUI, but we now want to add component power to it! What we do is shown on the image to the right: add wrappers around the original object classes (C3D*) and instead of linking the classes together by setting fields through the initial methods, do that in the binding methods. In other words, we have to spot exactly where C3DRenderingEngine, C3DUser and C3DDispatcher are used by a class other than itself, and make theses references component bindings. Of course, we also have to expose the interfaces that we are goining to use, hence the Dispatcher, Engine and User interface that have to be implemented.

19.4.3. How the application is written

First of all, have a look at the doc on C3D [c3d.xml] to remember how this application is written. Most important is the class diagram, showing C3DUser, C3DDispatcher and C3DRederingEngine. We decided that the only objects worth wrapping in components were those three. The rest is too small to be worth the hassle.

19.4.3.1. Creating the interfaces

What we need to do is to extract the interfaces of the Objects, ie the which methods are going to be called on the components. This means find out what methods are called from outside the Active Object. You can do that by searching in the classes where the calls are made on active objects. For this, **you have to know in detail which classes are going to be turned into component**. We have done that, and those exposed methods are put in the interfaces User, Engine and Dispatcher.

Tricky part: whatever way you look at components, you'll have to modify the initial code if these interfaces were not created at first go. You have to replace all the classes by their interface, when you use them in other files. If we had not already used interfaces in the C3D Objecct code, we would have had to replace all occurrences of C3DDispatcher by occurrences of Dispatcher.

Why do we have to do that, replacing classes by interfaces? That's due to the way components work. When the components are going to be bound, you're not binding the class themselves, but proxies to these classes. And these proxies implement the interfaces, and do not extend the classes.

19.4.3.2. Creating the Component Wrappers

You now have to create a class that englobes the previous Active Objects, and which is a composent representing the same functionality. How do you do that? Pretty simple. All you need to do is extend the Active Object class, and add to it the non-functional interfaces which go with the component. You have the binding interfaces to create, which basically say how to put together two Components, tell who is already attached, and how to separate them. These are the `lookupFc`, `bindFc`, `unbindFc`, and `listFc` methods.

This has been done in the *Impl files. Have a peek, for example, at UserImpl.java [../doc/ProActive_src_html/org.objectweb.proactive
What you have here are those component methods. Be even more careful with this `bindFc` method. In fact, it really binds the protected Dispatcher variable `c3ddispatcher`. This way, the C3DUser code can now use this varaible as if it was addressing the real Active Object. Just to be precise, we have to point out that you're going through proxies before reaching the Component, then the Active Object. But this is hidden by the ProActive layer, all you should know is you're addressing a Dispatcher, and you're fine!

19.4.3.3. Discarding direct reference acknowledgment

If you're out of luck, the code contains instructions to retain references to objects that call methods on the current Object. These methods have a signature ressembling `method(..., ActiveObject ao, ...)`. This is called, in ProActive, with a `ProActive.getStubOnThis()` (if you don't, and instead use 'this', the code won't work correctly on remote hosts!). If the local object uses this `ProActive.getStubOnThis()`, you're going to have trouble with components. The problem is that this design does not fit the component paradigm : you should be using declared interfaces bound with the bind methods, not be passing along references to self. So you have to remove these from the code, and make it component-oriented. But remember, **you should be using bind methods to attach other components**.

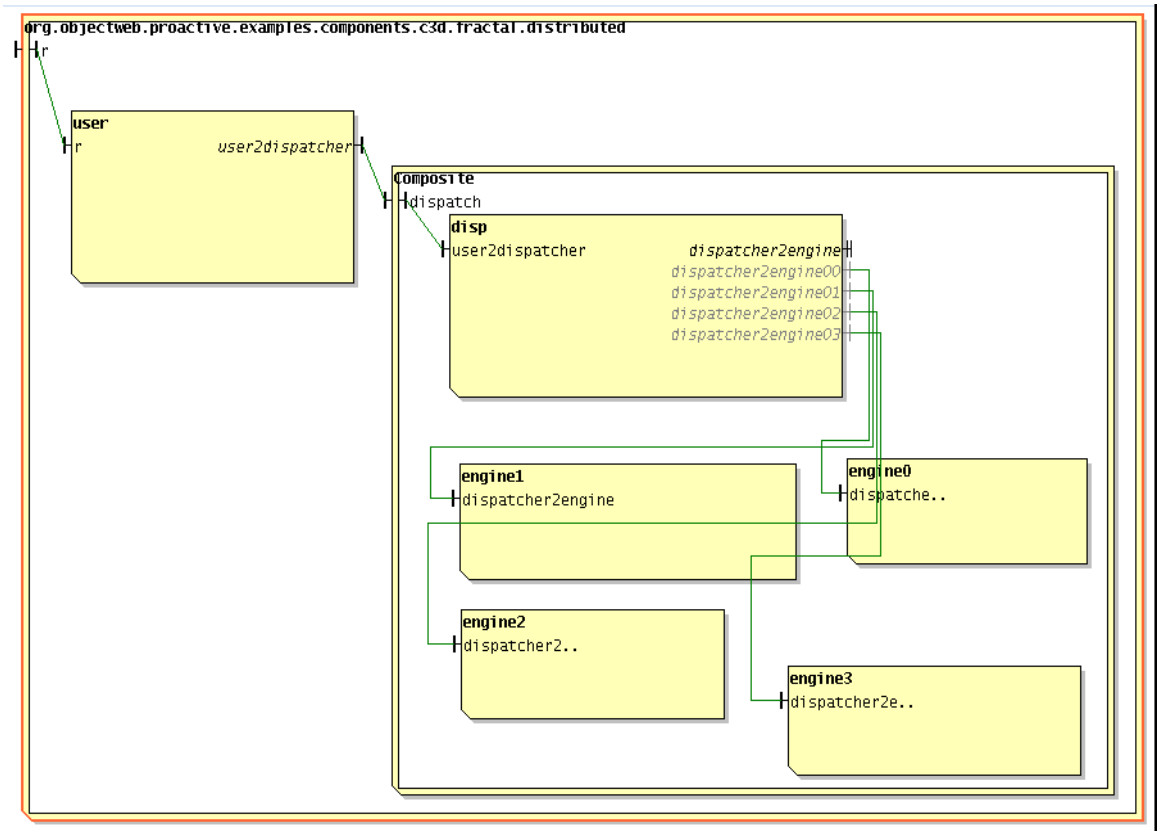
19.4.4. ADL

You may be wanting to see how we have bound the components together, now. Since the design is pretty simple, there is not much to it. We have used the fractal ADL, to avoid hard-coding bindings. So all of the information here is in the `components/c3d/fractal/` directory. There are the components (which interfaces they propose), and a "distributed.fractal" file, which is where the bindings are made. It includes the creation of a Composite component, just for the fun. You may want to explore it with the Fractal GUI provided with IC2D, it's easier to understand graphically. Here's the code, nevertheless, for you curiosity :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">
<definition
name="org.objectweb.proactive.examples.components.c3d.fractal.distributed">
  <interface signature="java.lang.Runnable" role="server" name="r"/>
  <component
definition="org.objectweb.proactive.examples.components.c3d.fractal.UserImpl" name="user" />
  <component name="Composite">
    <interface signature="org.objectweb.proactive.examples.c3d.Dispatcher"
role="server" name="dispatch"/>
    <component
definition="org.objectweb.proactive.examples.components.c3d.fractal.EngineImpl" name="engine2"/>
    <component
definition="org.objectweb.proactive.examples.components.c3d.fractal.EngineImpl" name="engine1"/>
    <component
definition="org.objectweb.proactive.examples.components.c3d.fractal.DispatcherImpl" name="disp"/>
    <binding client="this.dispatch" server="disp.user2dispatcher"/>
    <binding client="disp.dispatcher2engine00"
server="engine1.dispatcher2engine"/>
    <binding client="disp.dispatcher2engine01"
server="engine2.dispatcher2engine"/>
  </component>
  <binding client="this.r" server="user.r"/>
  <binding client="user.user2dispatcher" server="Composite.dispatch"/>
  <controller desc="composite"/>
  <coordinates color="-73" y0="0.11" x1="0.30" y1="0.33" name="user"
x0="0.03"/>
  <coordinates color="-73" y0="0.18" x1="0.99" y1="0.98" name="Composite"
x0="0.32">
    <coordinates color="-73" y0="0.60" x1="0.84" y1="0.89" name="engine2"
x0="0.23"/>
    <coordinates color="-73" y0="0.15" x1="0.99" y1="0.53" name="engine1"
x0="0.72"/>
    <coordinates color="-73" y0="0.12" x1="0.67" y1="0.42" name="disp"
x0="0.09"/>
  </coordinates>
</definition>
```

Here’s what it looks like when you explore it through the IC2D Component explorer



19.4.5. Source Code

You may find the source code of this application at the following locations :

- Active Object version [../doc/ProActive_src_html/org.objectweb.proactive.examples.c3d.index.xml] (examples/c3d)
- Component version [../doc/ProActive_src_html/org.objectweb.proactive.examples.components.c3d.index.xml] (examples/components/c3d)

Chapter 20. Component perspectives : a support for our research work

Currently, we have a functional implementation of the Fractal API and model. One can configure and deploy a distributed system made of components.

20.1. Optimizations

However, there are currently no optimizations apart from those already offered by ProActive, such as direct communications when communicating entities are in the same virtual machine.

We are studying strategies for optimizations that would allow us to reduce the interceptions and the network latency times (when the components are distributed). We have already implemented a first mechanism for shortcuts, which does not allow reconfiguration. We are currently working on a more sophisticated version that will allow dynamic reconfigurations.

20.2. Packaging

Reusability and Components should be accessible through predefined / preconfigured packages. A bit like enterprise archives for Enterprise JavaBeans, though there is also a notion of composition of deployment that should be addressed.

20.3. Graphical user interface

Another area of investigation is the tools for configuring, deploying and monitoring distributed component systems.

Because component based programming is somewhat analogous to the assembly of building blocks into a functional product, graphical tools are well suited for the design and monitoring of component based systems. The Fractal community actually proposes such a tool : the Fractal GUI. We have extended this tool to evaluate the feasibility of a full-fledge graphical interface for the design and monitoring of distributed components. The result is available within the IC2D GUI, you can try it out, but consider it as a product in alpha state. Development is indeed currently discontinued as we are waiting for a new release of the Fractal GUI, and some features are only partially implemented (runtime monitoring, composition of virtual nodes).

The GUI allows the creation of ADL files representing component systems, and - the other way around - also allows to load ADL files and get a visual representation of systems described in the ADL files. We have worked on the manipulation of virtual nodes - a deployment abstraction - : components display the virtual nodes where they are deployed, and it is also possible to compose virtual nodes

Ultimately, we would like to couple the visualization of components at runtime (currently unavailable here) with the standard monitoring capabilities of IC2D : we would get a structural view of the application in the Fractal GUI, and a topological view in the standard IC2D frame

20.3.1. Usage

If you want to try out the extended Fractal GUI for ProActive :

- start IC2D
- Components --> start components GUI
- to load an ADL file :

1. File --> Storage --> select the storage repository which is the root repository of your ADL files. *For example you can select the "src" directory of the ProActive distribution*

2. File --> Open --> select an ADL file in the storage repository. *For example you can select the "helloworld-distributed-wrappers.fractal" file in the src/org/objectweb/proactive/examples/components/helloworld directory of the ProActive distribution.*

- to modify an ADL file, you can use the Graph tab for a structural view, while the Dialog tab gives you access to the properties of the components, including the composition of the virtual ndoes.
- to save an ADL file : File --> Save

20.4. Other

Other areas of research that we are opening around this work include :

- wrapping legacy codes (MPI for instance) for interoperability with existing software
- behavioral studies with asynchronous distributed components
- formalism (ProActive is based on a formal deterministic model for asynchronous distributed objects)
- patterns for automatic and parameterizable configurations of component systems
- MxN data redistribution : automatic redistribution of data from M components to N components

20.5. Limitations

Some features of the Fractal model are not implemented :

- Shared components
- Templates (generic factories)

Custom controllers cannot yet be specified in the ADL.

Part V. Advanced

Table of Contents

Chapter 21. ProActive Peer-to-Peer Infrastructure

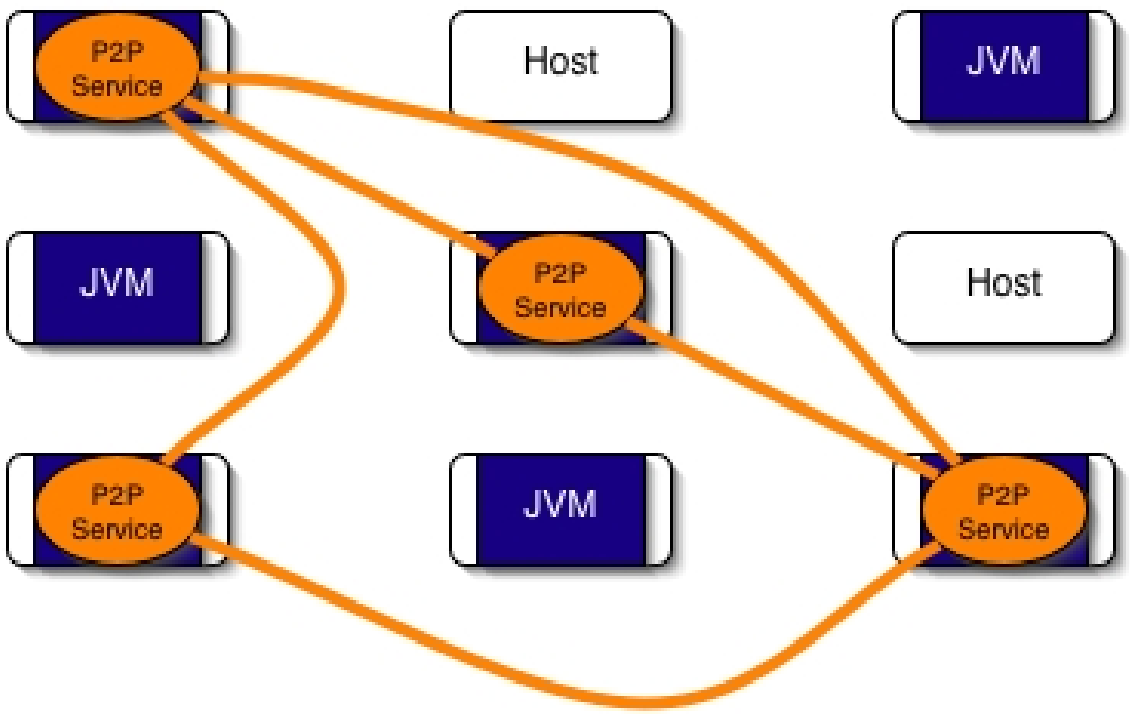
21.1. Overview

Computational Peer-To-Peer (P2P) is becoming a key execution environment. The potential of 100,000 nodes interconnected to execute a single application is rather appealing, especially for Grid computing. Mimicking data P2P, one could start a computation that no failure would ever be able to stop (and maybe nobody).

The ProActive P2P aims to use spare CPU cycles from organization's or institution's desktop workstations.

This short document explains how to create a simple computational P2P network. This network is a **dynamic JVMs network** which works like computational nodes.

The P2P infrastructure works as an overlay network. It works with a **P2P Service** which is a peer which in turn is in computational node. The P2P Service is implemented with a ProActive JVM and few Active Objects. The next figure shows an example of a network of hosts where some JVMs are running and several of them are running the P2P Service.



Example of a ProActive P2P infrastructure.

When the P2P infrastructure is running, it is very easy to obtain some nodes (JVMs). The next section describes how to use it.

Further research information is available here [http://www-sop.inria.fr/oasis/personnel/Alexandre.Di_Costanzo/publications.xml].

21.2. The P2P Infrastructure Model

The goals of this work are to use sparse CPU cycles from institutions' desktop workstations combined with grids and clusters. Desktop workstations are not available all the time for sharing computation times with different users

other than the workstation owner. Grids and clusters have the same problem as normal users don't want to share their usage time.

Managing different sorts of resources (grids, clusters, desktop workstations) as a single network of resources with a high instability between them needs a fully decentralized and dynamic approach.

Therefore, P2P is a good solution for sharing a dynamic JVM network, where JVMs are the shared resources. Thereby, the P2P Infrastructure is a P2P network which shares JVMs for computation. This infrastructure is completely self-organized and fully configurable.

Before going on to consider the P2P infrastructure, it's important to define what Peer-to-Peer is.

21.2.1. What is Peer-to-Peer?

There are a lot of P2P definitions, many of them are similar to other distributed infrastructures, such as Grid, client / server, etc. There are 2 better definitions which describe really P2P well:

- From Peer-to-Peer Harnessing the Power of Disruptive Technologies (edited by Andy Oram):

"[...] P2P is a class of applications that take advantage of **resources** - available at the edges of the Internet [...]"

- And from A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications (Rdiger Schollmeier - P2P'01):

"[...] Peers are **accessible** by other peers directly [...] Any arbitrary chosen peer can be **removed** from the network **without fault** [...]"

P2P's focus on sharing, decentralization, instability and fault tolerance.

21.2.2. The P2P Infrastructure in short

21.2.2.1. Bootstrapping: First Contact

A fresh (or new) peer which would like to join the P2P network, will encounter a serious bootstrapping problem or first contact problem: "How can it connect to the P2P network?"

A solution for that is to use a specific protocol. ProActive provides an interface for a network-centric services protocol which is named JINI. JINI can be used for discovering services in a dynamic computing environment, such as a fresh peer which would like to join a P2P network. This protocol is perfectly adapted to solve the bootstrapping problem. However, there is a serious drawback for using a protocol such as JINI as peer discovering protocol. JINI is limited to working only in the same sub-network. That means JINI doesn't pass through firewalls or NAT and can't be considered to be used for Internet.

Therefore, a different solution for the bootstrapping problem was chosen. The solution for ProActive first contact P2P is inspired from Data P2P Networks. This solution is based on real life , i.e. when a person wants to join a community, this person has to first know another person who is already a member of the community. After the first person has contacted the community member, the new person is introduced to all the community members.

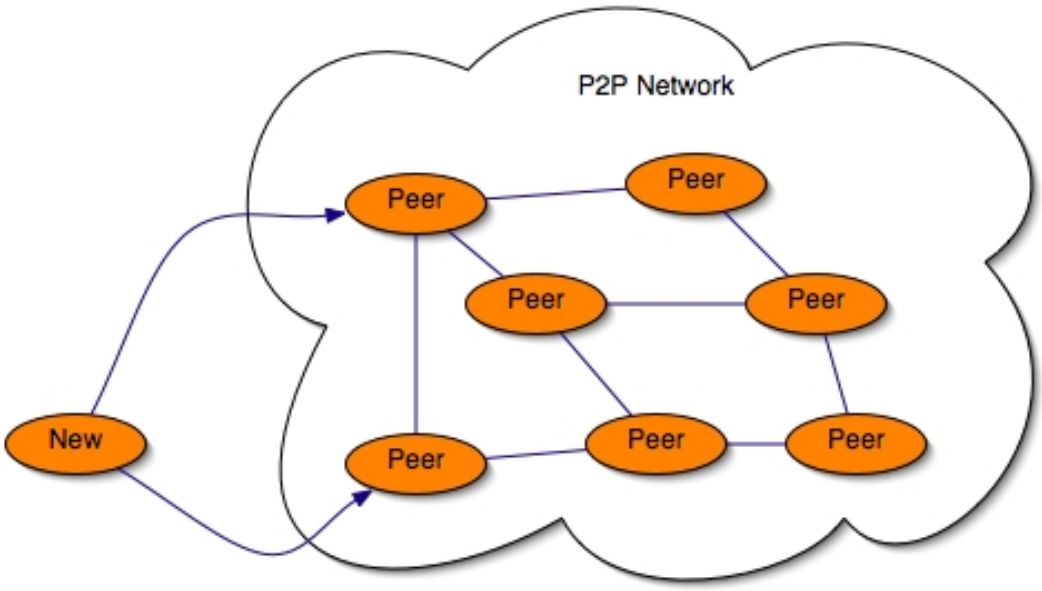
The ProActive P2P bootstrapping protocol works as follows:

- A fresh peer has a list of "server" addresses. These are peers which have a high potential to be available and to be in the P2P network, they are in a certain way the P2P network core.
- With this list the fresh peer tries to contact each server. When a server is reached the server is added to ithe fresh peer's list of known peers (acquaintances).

- Then the fresh peer knows some servers, it is in the P2P Network and it is no longer a fresh peer, it is a peer of the P2P network.

Furthermore, in the case of the fresh peer not able to contact any servers from the list, the fresh peer will try every TTU (see below, about Time To Update parameter) to re-contact all of them until one or several of them are finally available. At any moment when the peer knows nobody because all of its acquaintances are no longer available, the peer will try to contact all the servers as explained earlier.

An example of a fresh peer which is trying to join a P2P network is shown by the next Figure. The new peer has 2 servers to contact in order to join the existing P2P infrastructure.



Example of first contact (Bootstrapping).

21.2.2.2. Discovering and Self-Organizing in Continue

The main particularity of a P2P network is the peers high volatility. This results from various attributes which compose P2P:

- Peers run on different kinds of computers: desktop workstations, laptops, servers, cluster nodes, etc.
- Each peer has a particular configuration: operating system, etc.
- Communicating network between peers consists of different speed connections: modem, 100Mb Ethernet, fiber channel, etc.
- Peers are not available all the time and not all at the same moment.
- Peer latency is not equal for all.
- etc.

The result is the instability of the P2P network. But the ProActive P2P infrastructure deals with these problems with transparency.

ProActive P2P infrastructure aims to maintain a created P2P network alive while there are available peers in the network, this is called self-organizing of the P2P network. Because P2P doesn't have exterior entities, such as centralized servers which maintain peer data bases, the P2P network has to be self-organized. That means all peers should be enabled to stay in the P2P network by their own means.

There is a solution which is widely used in data P2P networks; this consists of each peer keeping a list of their neighbors, a peer's neighbor is typically a peer close to it (IP address or geographically).

In the same way, this idea was selected to keep the ProActive P2P infrastructure up. All peers have to maintain a list of **acquaintances**. At the beginning, when a fresh peer has just joined the P2P infrastructure, it knows only peers from its bootstrapping step. However, depending on how long the list of servers is, many of them could be unreachable, unavailable, etc. and the fresh peer ends up knowing a small number of acquaintances. Knowing a small number of acquaintances is a real problem in a dynamic P2P network when all the servers will be unavailable, the fresh peer will be unconnected from the P2P infrastructure.

Therefore, the ProActive P2P infrastructure uses a specific parameter called: **Number Of Acquaintances (NOA)**. This is a minimum size of the list of acquaintances of all peers. The more the peers are highly dynamic, the more NOA should be high. Thereby, a peer must discover new acquaintances through the P2P infrastructure.

In the next section, we will see in detail how the message protocol works. For the moment we will just explain briefly the discovering acquaintances process without going into detail about the message protocol.

The peer called "Alice" has 2 acquaintances resulting from its first contact with the P2P infrastructure and by default NOA is 10 peers. Alice must find at least 8 peers to be able to stay with a certain guarantee inside the infrastructure.

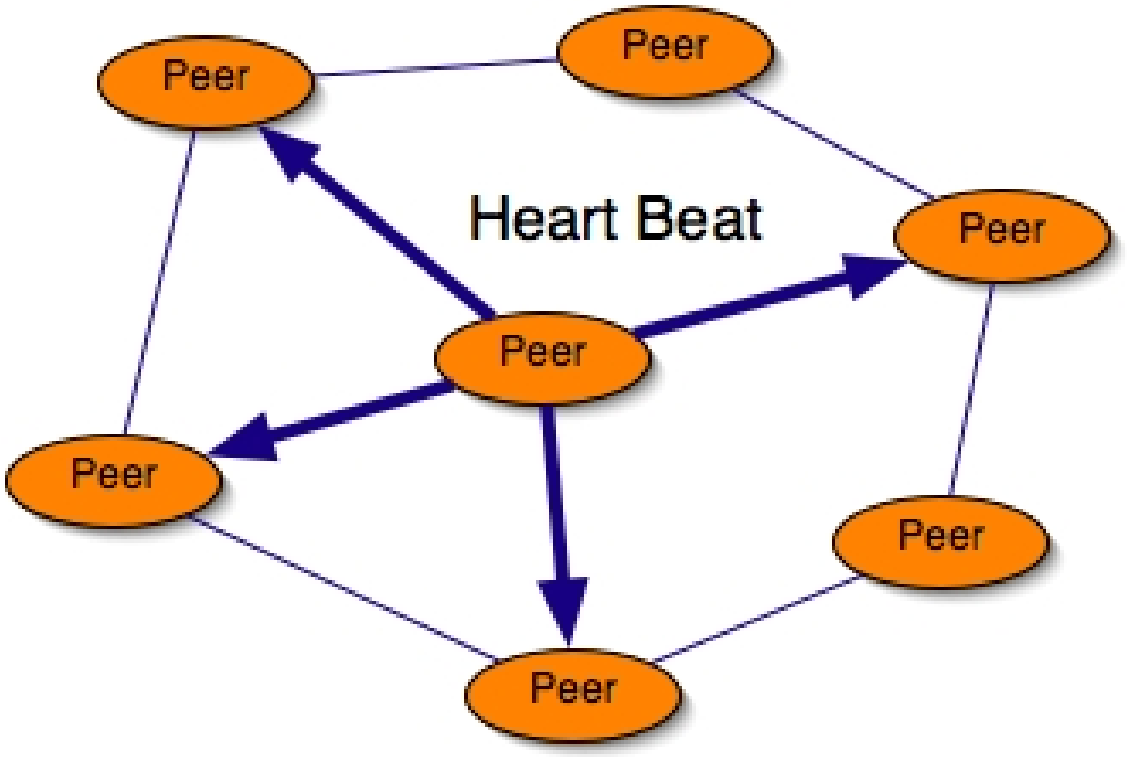
The acquaintance discovering works as follows:

- Send an exploring message to all of its acquaintances, and wait for responses from new acquaintances (not peers that have already been contacted peers and not already known peers).
- When receiving an exploring message:
    - Forward the message to acquaintances until the message Time To Live (TTL) reaches 0.
    - Choose to be or not to be an acquaintance of the asking peer.

In order to not have isolated peers in the infrastructure, all peers registration are symmetric. That means if Alice knows the peer "Bob", Bod also knows Alice. Hence, when a peer chooses whether to be an acquaintance or not, the peer has to check previously in its own acquaintance list if it doesn't already know the asking peer. Next, if it's an unknown peer, the peer decides with a random function to be an acquaintance or not. With the parameter of **agree responses**, it is possible to configure the percentage of positive responses to an exploring message. The random function is a temporary solution to solve the flooding problem due to the message protocol (see next section), we are thinking of using a new parameter Maximum Number of Acquaintances and improving the message protocol. For the moment, we don't consider peers IP addresses or geographical location of the peers as an acquaintances criteria.

As the P2P infrastructure is a dynamic environment, the list of acquaintances must also be dynamic. Many acquaintances could be unavailable and must be removed of the list. When the size of the list is less than the NOA, the peer has to discover new peers. Therefore, all peers keep their lists up-to-date. That's why a new parameter must be introduced: **Time To Update (TTU)**. The peer must frequency check its own acquaintances' list to remove unavailable peers and discover new peers. To verify the acquaintances availability, the peer send a **Heart Beat** to all of its acquaintances. The heart beat is sent every TTU.

The next figure shows a peer which is sending a heart beat to all of its acquaintances.



Heart beat sent every TTU.

21.2.2.3. Asking Computational Nodes

The main goal of this work is to provide an infrastructure for sharing computational nodes (JVMs). Therefore, a resource query mechanism is needed; there are 2 types of resources in this context, thus 2 query types:

- Exploring the P2P infrastructure to search new acquaintances.
- Asking free computational nodes to deploy distributed applications.

The mechanism is similar to Gnutella's communication system: **Breadth-First Search** algorithm (BFS). The system is message-based with application-level routing.

All BFS messages must contain this information:

- A Unique Universal Message Identifier (UUID): this message identifier is not totally universally unique, it is just unique for the infrastructure;
- The **Time To Live (TTL) infrastructure parameter**, in number of hops;
- A reference to the requester peer. The peer waits for responses for nodes or acquaintances.

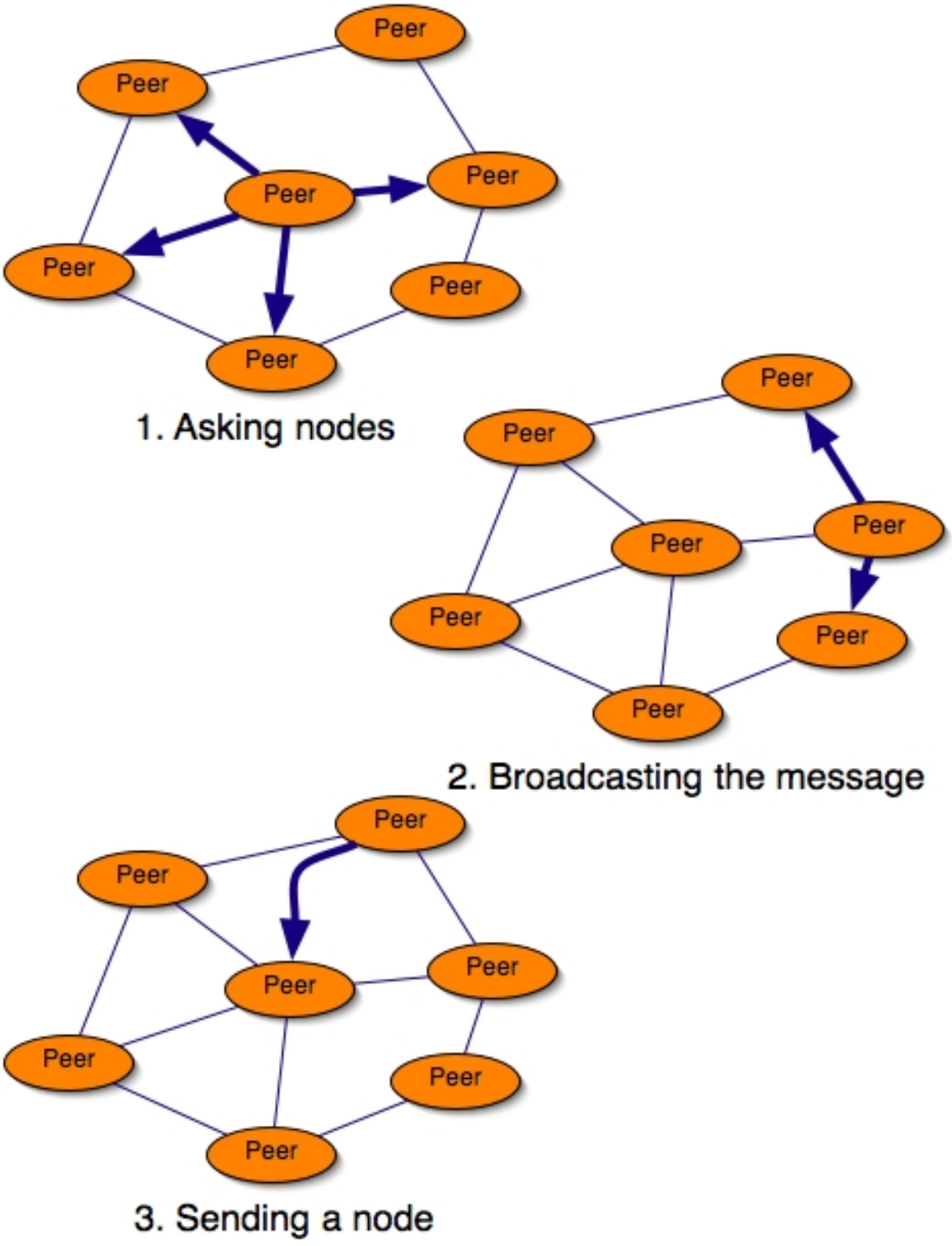
Our BFS inspired version works as follow:

- **Broadcasting** a request message to all of its acquaintances with an **UUID**, and **TTL**, and **number of asked nodes**.
- When **receiving** a message:

- Test the message UUID, **is it an old message?**
  - Yes, it is: continue;
  - No, it’s not:
    - **Keep** the **UUID**;
    - I have a free node:
    -
      - Send the node reference to the caller and waiting an **ACK** until **timeout**
      - if **timeout** is reached or **NACK**
      -
        - continue;
      - if **ACK** and **asked nodes - 1 > 0** and **TTL > 0** then
      -
        - **Broadcast** with **TTL - 1** and **asked nodes -1**
      - continue;

Gnutella’s BFS got a lot of justified critics for scaling, bandwidth, etc. It is true this protocol is not good enough but we’re working to improve it. We are inquiring into solutions with a not fixed TTL to avoid network flooding.

The next Figure shows briefly the execution of the inspired BFS algorithm:



Asking nodes to acquaintances and getting a node.

21.3. The P2P Infrastructure Implementation

21.3.1. Peers Implementation

The P2P infrastructure is implemented with ProActive. Thus the shared resource is not a JVMs but a ProActive node, nodes are like a container which receives work.

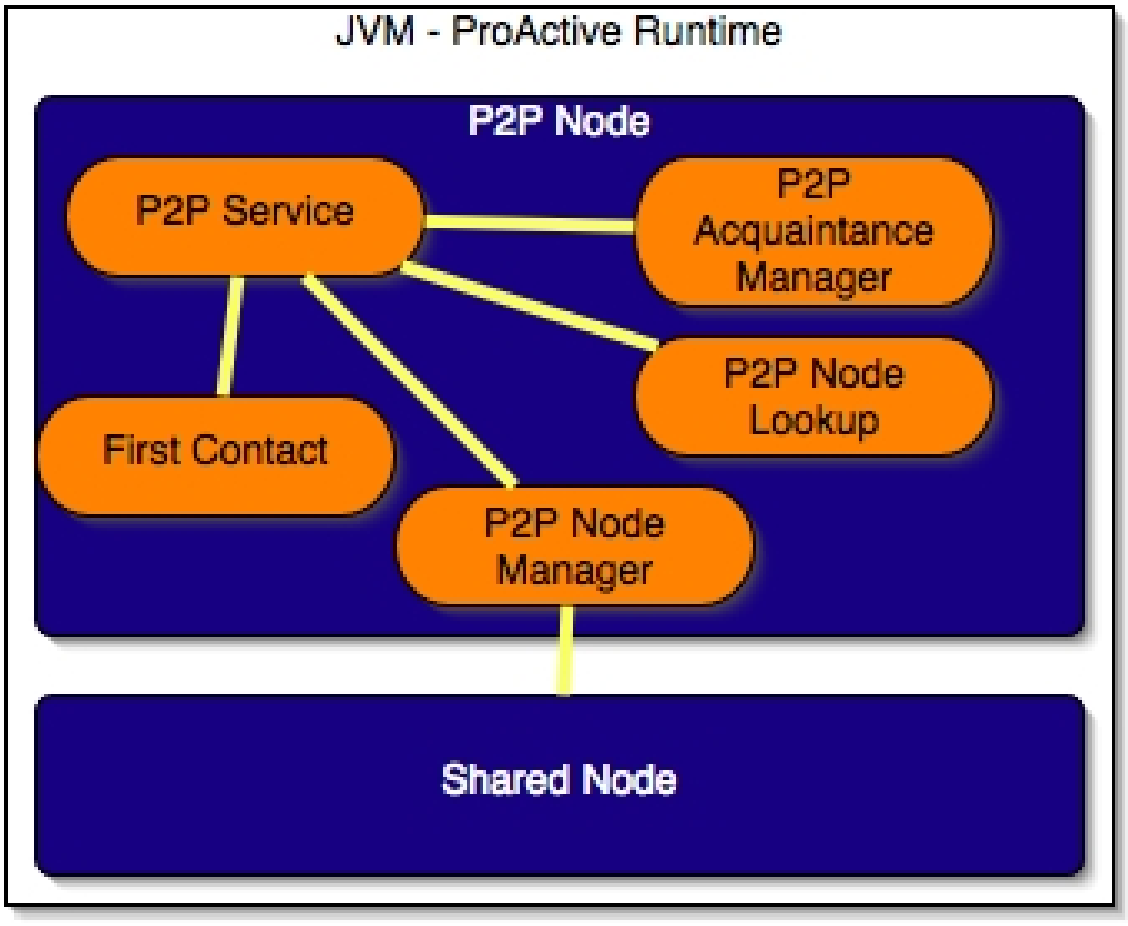
The P2P infrastructure is not directly implemented in the ProActive core at the ProActive runtime level because we choose to be above communication protocols, such as RMI, HTTP, Ibis, etc. Therefore, the P2P infrastructure can use RMI or HTTP as communication layer. Hence, the P2P infrastructure is implemented with classic ProActive active objects and especially with ProActive typed group for broadcasting communications between peers due to your inspired BFS.

Using active objects for the implementation is a good mapping with the idea of a peer which is an independent entities that works as a server with a FIFO request queue. The peer is also a client which sends requests to other peers.

The list of P2P active objects:

- **P2PService** : is the main active object. It serves all register requests or resource queries, such as nodes or acquaintances.
- **P2PNodeManager** : works together with the P2PService, this active object manages one or several shared nodes. It handles the booking node system, see here for more details.
- **P2PAcquaintanceManager** : manages the list of acquaintances and provides group communication, see next section.
- **P2PNodeLookup** : works as a broker when the P2PService asks nodes. All the asking node protocol is inside it. This broker can migrate to a different node to be closer to the deployed application.
- **FirstContact** : it's the bootstrapping object.

The Figure below shows the connection between all active objects:



Nodes and Active Objects which make up a P2P Service.

All communications between peers use Group communication but for sending a response to a request message, it's a point-to-point communication. Though ProActive communications are asynchronous, it's not really messages which are sent between peers. Nevertheless, it's not a real problem; ProActive is implemented above Java RMI which is RPC and RPC is synchronous. However, ProActive uses future mechanism and Rendez-vous method to turn RPC methods to asynchronous. That means ProActive is asynchronous RPC. Rendez-vous is interesting in your case because it guarantees the method is successfully received by the receiver. With the Heart beat message which is sent a Java exception when an acquaintance is down.

The P2PAcquaintanceManager manages the list of acquaintances, this list is represented by a ProActive typed group of P2PService. This is the point of the next section.

21.3.2. Dynamic Shared ProActive Group

ProActive typed group does not allow access to group elements and make calls from different active objects to the same group is not possible, i.e. a group can not be shared. However, the point of the P2P infrastructure is to broadcast messages to all members on the acquaintance list, ProActive typed group is perfect for doing that. A typed group of P2PService is a good implementation of the acquaintance list design.

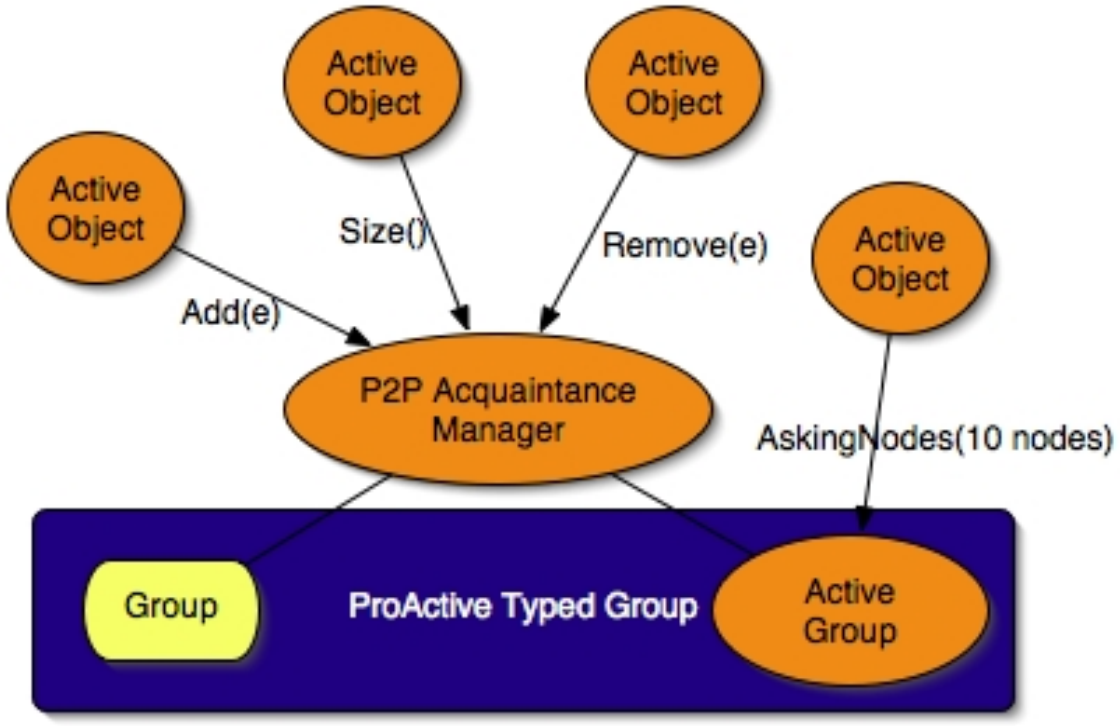
But a typed group does not support to be shared by many active objects, especially for making group method calls from different objects, adding / removing / etc. members in the group. For the P2P infrastructure the P2PAcquaintanceManager (PAM) was designed.

The PAM is a standard active object, at its initialization it constructs an empty P2PService group. The PAM provides an access to few group methods, such as removing, adding and group size methods. All other active objects, such as P2PService or P2PNodeLookup, have to use PAM methods to access the group. The PAM works as a server with an FIFO queue behind the group.

That solves the problem of group members accessing but not how other active objects can call methods on the group. The ProActive group API provides a method to active a group that is made possible to get ProActive reference on the group. The PAM actives the group after its creation. P2PService, P2PNodeLookup and all get the group reference from a PAM's getter.

The PAM, during its activity, frequently sends heart beats to remove unavailable peers. The P2PService adds, via the PAM, new discovered acquaintances (P2PService) and the P2PNodeLookup calls group methods to ask nodes to the group reference. The P2PService does also group method calls.

In short, this can be seen in the next Figure:



Dynamic Shared ProActive Typed Group.

We just explained how to share a typed group between active objects but that is not solve all the problems. For the moment, the BFS implementation with broadcasting to all acquaintances each time is not perfect due to the message which is always send back to the previous sender. We are working to add member exclusion in a group method call.

21.3.3. Sharing Node Mechanism

The sharing node mechanism is an independent activity from the P2P service. Nodes are the sharing resource of this P2P network. This activity is handled by the P2PNodeManager active object.

At the initialization of the P2PNodeManager (PNM), it has to instantiate the shared resource. By default, it's 1 ProActive nodes by CPUs, for example on a single processor machine the PNM starts 1 node and on a bi-processors machine it starts 2 nodes. It's possible to choose to share only a single node. An another way is to share nodes from an XML deployment descriptor file by specifing the descriptor to the PNM which actives the deployment and gets nodes ready to share.

When the P2P service receives a node request, the request is forwarded (after the BFS broadcast) to the PNM which checks for a free node. In the case of at least 1 free node, the PNM must book the node and send back a reference to the the node to the original request sender. However, the booking remains valid for a predetermined time, this time expires after a configurable timeout. The PNM knows if the node is used or not by testing the active object presence inside the node. Consequently, at the end of the booking time, the PNM kills the node, the node is no longer usable. Though, some applications need empty nodes for a long time before using them, thereby there is a pseudo expand booking time system: creating "Dummy" active objects in booked nodes for later use. This system is allowed by the P2PNodeLookup.

The P2PNodeLookup could receive more nodes than it needs, for all additional nodes, the P2PNodeLookup sends a message to all PNMs' nodes to cancel its booking on the node.

The deployed applications have to leave nodes after use. Therefore, the PNM offers a leaving node mechanism that is the application sent a leaving message for a specified node to the PNM which kills all node's active objects by terminating their bodies and kills the node. After that, the PNM creates a new node which is ready for sharing.

However, if nodes are deployed by an XML descriptor the PNM does't kill the node, it just terminates all its active objects and re-shares the same node.

The asking node mechanism is allowed by the P2PNodeLookup, this object is active by the P2PService when it receives an asking node request from an application. The P2PNodeLookup (PNL) works as a broker, it could migrate to another place (node, machine, etc.) to be near the application.

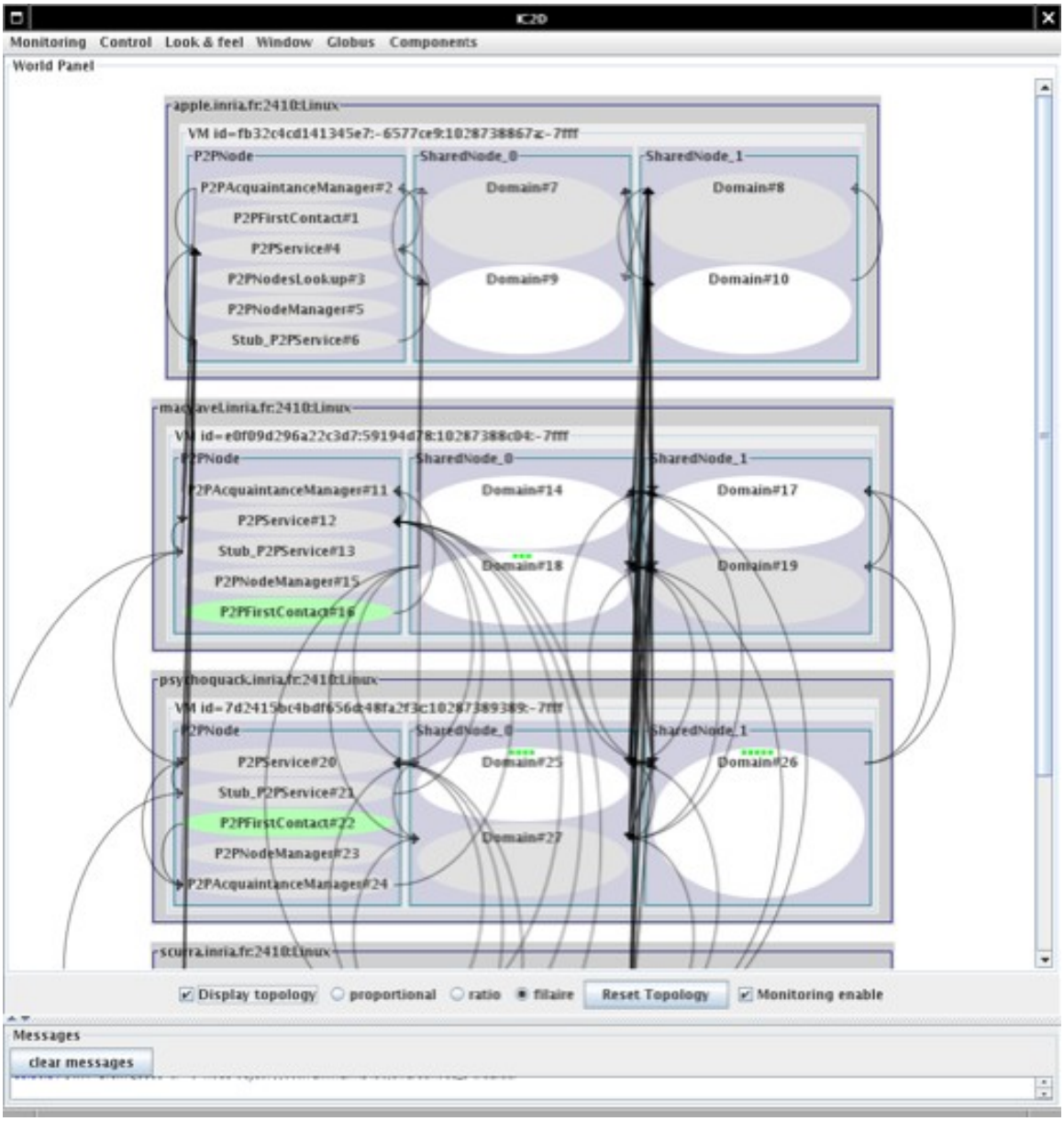
The PNL aims to find the number of nodes requested by the application. It uses the BFS to frequently flood the network until it gets all nodes or until the timeout is reached. However, the application can ask to the maximum number of nodes, in that case the PNL asks to nodes until the end of the application. The PNL provides a listener / producer event mechanism which is great for the application which wants to know when a node is found.

Finally, the application kills nodes by the PNL which is in charge of contacting all the PNMs of each node and asks them to leave nodes. The PNMs leave nodes with the same mechanism of the booking timeout.

Lastly, the asking nodes mechanism with the PNL is fully integrated to the ProActive XML deployment descriptor.

21.3.4. IC2D Screen shot

A screen shot made with IC2D. You can see 3 P2P services which are sharing 2 nodes (bi-processors machines). Inside the nodes there are some active Domain objects from the nBody application which is deployed on this small P2P infrastructure.



nBody application deployed on P2P Infrastructure.

21.4. Installing and Using the P2P Infrastructure

21.4.1. Create your P2P Network

The P2P infrastructure is self-organized and configurable. When the infrastructure is running you have nothing to do to keep it up. There are 3 main parameters to configure:

- **Time To Update (TTU)** : each peer checks if its known peers are available when TTU expires. By default, its value is 1 minute.
- **Number Of Acquaintances (NOA)** : is the minimal number of peers one peer needs to know to keep up the infrastructure. By default, its value is 10 peers.

- **Time To Live (TTL)** : in hops for JVMs (node) depth search (acquisition). By default, its value is 5 hops.

All parameter descriptions and the way to change their default values are explained here [Configuration.xml#p2p_properties]. Next section shows how to configure the infrastructure when starting the P2P Service with the command line.

The bootstrapping or first contact problem is how a new peer can join the p2p infrastructure. We solved this problem by just specifying one or several addresses of supposed peers which are running in the p2p infrastructure. Next, we will explain how and where you can specify this list of peers.

Now, you just have to start peers. There are two ways to do so:

21.4.1.1. Quick Start Peer

This method explains how to rapidly launch a simple P2P Service on one host.

ProActive provides a very simple *script* to start a P2P Service on your local host. The name of this script is **startP2PService**.

- UNIX, GNU/Linux, BSD and MacOsX systems: the script is located in **ProActive/scripts/unix/p2p/startP2PService.sh** file.
- Microsoft Windows system: the script is located in **ProActive/p2p/scripts/windows/p2p/startP2PService.bat** file.

Before launching this script, you have to specify some parameters to this command:

```
startP2PService [-acq acquisitionMethod] [-port portNumber] [-s Peer ...] [\
-f PeersListFile]
```

- **-acq acquisitionMethod** the ProActive Runtime communication protocol used. Examples: rmi, http, ibis, ... By default it is *rmi*.
- **-port portNumber** is the port number where the P2P Service will listen. By default it is *2410*
- **-s Peer ...** specify addresses of peers which are used to join the P2P infrastructure. Example:

  ```
  rmi://applepie.proactive.org:8080
  ```

- **-f PeersListFile** same of **-s** but peers are specified in file **ServerListFile**. One per line.

More options:

- **-noa NOA** in number of host. NOA is the minimal number of peers one peer needs to know to keep up the infrastructure. By default, its value is 10 peers.
- **-ttu TTU** is in minutes. Each peer sends a heart beat to its acquaintances. By default, its value is 1 minute.
- **-ttl TTL** is in hop. TTL represents live time messages in hops of JVMs (node). By default, its value is 5 hops.
- **-capacity Number_of_Messages** is the maximum memory size to stock message UUID. Default value is 1000 messages UUID.

- **-exploring Percentage** is the percentage of agree response when a peer is looking for acquaintances. By default, its value is 66%.
- **-booking Time** in ms it takes while booking a shared node. It's the maximum time in milliseconds to create at least an active object in the shared node. After this time, and if no active objects are created, the shared node will leave and the peer which gets this shared node will be no longer be able to use it. Default is 3 minutes.
- **-node_acq Time** in milliseconds which is the timeout for node acquisition. The default value is 3 minutes.
- **-lookup Time** is the lookup frequency in milliseconds for re-asking nodes. By default, it's value is 30 seconds.
- **-no_multi_proc_nodes** to share only a node. Otherwise, 1 node by CPU that means the p2p service which is running on a bi-pro will share 2 nodes. By default, 1 shared node for 1 CPU.
- **-xml_path** to share nodes from a XML deployment descriptor file. This option takes a file path. By default, no descriptors are specified. That means the P2P Service shares only one local node or one local node by CPUs.

All arguments are optional.

Comment: With the UNIX version of the startP2PService script, the P2P service is persistent and runs like a UNIX *nice* process. If the JVMs that are running the P2P service stop (for a Java exception) the script re-starts a new one.

21.4.1.2. Usage Example

In this illustration, we will explain how to start a first peer and then how new peers can create a P2P network with the first one.

Start the first peer with *rmi* protocol and listening on port *2410*:

```
first.peer.host$startP2PService.sh -acq rmi -port 2410
```

Now, start new peers and connect them to the first peer to create a tiny P2P network:

```
second.peer.host$startP2PService.sh -acq rmi -port 2410 -s rmi://first.peer\
.host
```

```
third.peer.host$startP2PService.sh -acq rmi -port 2602 -s rmi://first.peer.\
host
```

You could specify a different port number for each peer.

Use a file to specify the addresses of peers:

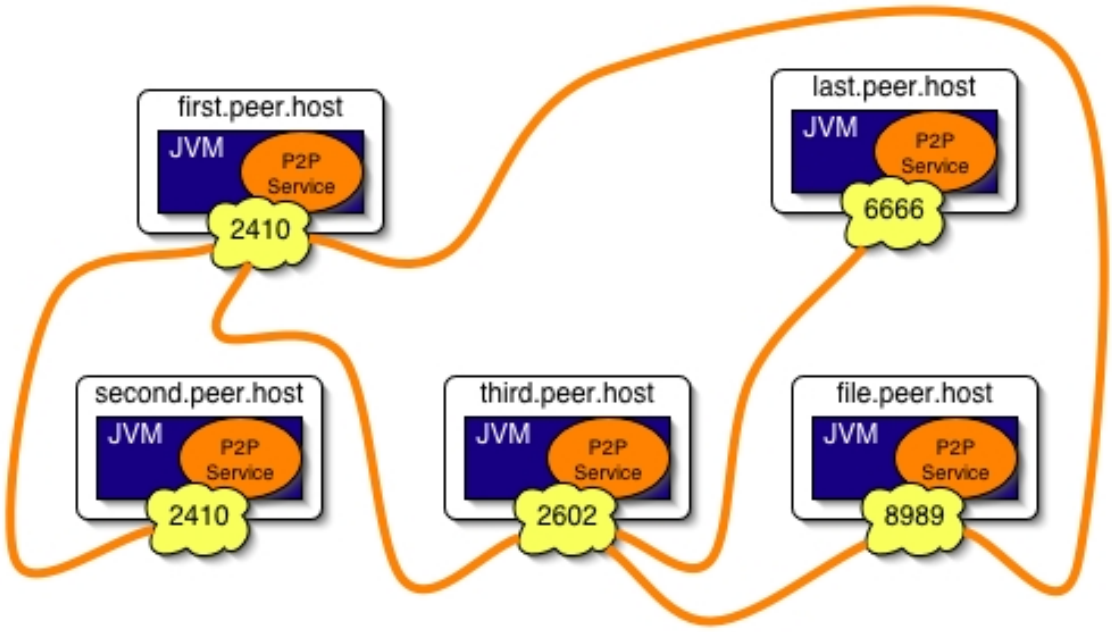
The file *hosts.file*:

```
rmi://first.peer.host:2410
rmi://third.peer.host:2602
```

```
file.peer.host$startP2PService.sh -acq rmi -port 8989 -f hosts.file
```

Lastly, a new peer joins the P2P network:

```
last.peer.host$startP2PService.sh -acq rmi -port 6666 -s rmi://third.peer.h\
ost:2410
```



Usage example P2P network (after firsts connections).

21.4.1.3. The P2P Daemon

The daemon aims to use computers in Peer-to-Peer computations. There will be a Java virtual machine sleeping on your computer and waking up at scheduled times to get some work done.

By default, the JVM is scheduled to wake up during the weekend and during the night. Next, we will explain how to change the schedule. The JVM is running with the lowest priority.

21.4.1.3.1. Installation

UNIX

Go to the directory: **ProActive/compile** and run this command:

```
$ ./build daemon
```

Before compiling you should change some parameters like the daemon user or the port in the file:

```
ProActive/p2p/src/common/proactivep2p.h
```

Ask your system administrator to add the daemon in a crontab or init.d. The process to run is located here:

```
ProActive/p2p/build/proactivep2p
```

Microsoft Windows

To compile daemon source (in c++), we don't provide any automatic script, you have to do it yourself. All sources for Windows are in the directory: **ProActive/p2p/src/windows**. If you use Microsoft Visual Studio, you can find in the src directory the Microsoft VS project files.

After that you are ready to install the daemon with Windows, you just have to run this script:

```
C:>ProActive\scripts\windows\p2p\Service\install.bat
```

To remove the daemon:

```
C:>ProActive\scripts\windows\p2p\Service\remove.bat
```

Comment: By default the port number of the daemon is **9015**.

21.4.1.3.2. Configuration

The daemon is configured with XML files in the **ProActive/p2p/config/** directory. To find the correct configuration file, the daemon will first try with a host dependent file: **config/proactivep2p.${HOST}.xml** for example: **config/proactivep2p.camel.inria.fr.xml** if the daemon is running on the host named **camel.inria.fr.**

If this host specific file is not found, the daemon will load **config/proactivep2p.xml**. This mechanism can be useful to setup a default configuration and have a specific configuration for some hosts.

The reference is the XML Schema in proactivep2p.xsd [p2p_files/proactivep2p.xsd]. For those not fluent in XML Schema, here is a description of all markup tags--

The root element in **<configFile>** it contains one or many **<p2pconfig>** . This latter element can start with a **<loadconfig path="path/to/xml"/>** it will include the designated XML file. After these file inclusions, you can with **<host name="name.domain">** specify which hosts are concerned by the configuration. Then there can be a **<configForHost>** element containing a configuration for the selected hosts and/or a **<default>** element if no suitable configuration was already found.

Bear in mind that the XML parser sees a lot of configuration and the first that matches is used and the parsing is finished. This means that the elements we have just seen are tightly linked together. For example if an XML file designated by a **<loadconfig>** contains a **<default>** element, then after this file no other element will be evaluated. This is because either a configuration was already found so the parsing stops, or no configuration matched and the **<default>** does, so the parsing ends.

The proper configuration is contained in a **<configForHost>** or **<default>** element. It consists of the scheduled times for work and the hosts where we register ourselves. Here is an example:

```
<periods>
 <period>
  <start day="monday" hour="18"
minute="0"/>
  <end day="tuesday" hour="6"
minute="0"/>
  </period>
  <period>
   <start day="saturday" hour="0"
minute="0"/>
   <end day="monday" hour="6"
minute="0"/>
  </period>
</periods>
<register>
 <registry url="trinidad.inria.fr"/>
 <registry url="amda.inria.fr"/>
 <registry url="tranquility.inria.fr"/>
 <registry url="psychoquack.inria.fr"/>
</register>
```

In this example we clearly see that the JVM will wake up Monday evening and shut down Tuesday morning. It will also work during the weekend. In the **<register>** part we put the URL in which we will register ourselves, in the example we used the short form which is equivalent to *rmi://host:9301*.

21.4.1.3.3. Control

The following commands only work with UNIX friendly systems.

- **Stop the JVM** : This command will stop the JVM and will restart it at the next scheduled time, which is the day after:

```
$ProActive/p2p/build/p2pctl stop [hostname]
```

- **Kill the daemon** :

```
$ProActive/p2p/build/p2pctl killdaemon [hostname]
```

- **Restart the daemon** :

```
$ProActive/p2p/build/p2pctl restart [hostname]
```

- **Test the daemon** :

```
$ProActive/p2p/build/p2pctl alive [hostname]
```

- **Flush the daemon logs** :

```
$ProActive/p2p/build/p2pctl flush [hostname]
```

hostname is the name of the remote host which the daemon command is sent to. This parameter is optional, if the host name is not specified the command is executed on the local host.

Under Windows you could use some littles scripts in **ProActive//script/windows/p2p/JVM** to do that.

All daemon logs are written in a file. All logs are available in:

```
ProActive/p2p/build/logs/hostname
```

21.4.2. Example of Acquiring Nodes by ProActive XML Deployment Descriptors

You can customize some P2P settings such as:

- **nodesAsked** is the number of nodes you want from the P2P infrastructure. Setting **MAX** as value is equivalent to an infinite number of nodes. This attribute is required.
- **acq** is the communication protocol that's used to communicate with this P2P Service. All ProActive communication protocols are supported: rmi, http, etc. Default is rmi.
- **port** represents the port number on which to start the P2P Service. Default is 2410. The port is used by the communication protocol.
- The **NOA Number Of Acquaintances** is the minimal number of peers one peer needs to know to keep up the infrastructure. By default, its value is 10 peers.
- The **TTU Time To Update** each peer sends a heart beat to its acquaintances. By default, its value is 1 minute.
- The **TTL Time To Live** represents messages live time in hops of JVMs (node). By default, its value is 5 hops.
- **multi_proc_nodes** is a boolean (use true or false) attribute. When its value is true the P2P service will share 1 node by CPU, if not only one node is shared. By default, its value is true, i.e. 1 node / CPU.
- **xml_path** is used with a XML deployment descriptor path. The P2P Service shares nodes which are deployed by the descriptor. No default nodes are shared.
- **booking_nodes** is a boolean value (true or false). During asking nodes processs there is a timeout, booking timeout is used for obtaining nodes. That means if no active objects are created before the end of the timeout, the node will be free and no longer shared. To avoid the booking timeout, put this attribute at true, obtained nodes will be permanently booked for you. By default, its value is false. See below, for more information about the booking timeout.

With elements **acq** and **port**, if a P2P Service is already running with this configuration the descriptor will use this one, if not a new one is started.

In order to get nodes, the **peerSet** tag will allow you to specify entry point of your P2P Infrastructure.

You can get nodes from the P2P Infrastructure using the ProActive Deployment Descriptor as described above.

In fact you will ask for a certain number of nodes and ProActive will notify a "listener" (one of your class), every time a new node is available.

```
 ProActiveDescriptor pad =
ProActive.getProactiveDescriptor("myP2PXmlDescrip\
tor.xml");
 // getting virtual node "p2pvn" defined in the ProActive Deployement Descrip\
tor
 VirtualNode vn = pad.getVirtualNode("p2pvn");

 // adding "this" or anyother class has a listener of the "NodeCreationEvent"
 ((VirtualNodeImpl) vn).addNodeCreationEventListener(this);
 //activate that virtual node
 vn.activate();
```

As you can see, the class executing this code must implement an interface in order to be notified when a new node is available from the P2P infrastructure.

Basically you will have to implement the interface NodeCreationEventListener that can be found in package org.objectweb.proactive.core.event. For example, this method will be called every time a new host is acquired:

```
 public void nodeCreated(NodeCreationEvent event)
 {
 // get the node
 Node newNode = event.getNode();
   // now you can create an active object on your node.
 }
```

You should carefully notice that you can be notified at any time, whatever the code you are executing, once you have activated the virtual node.

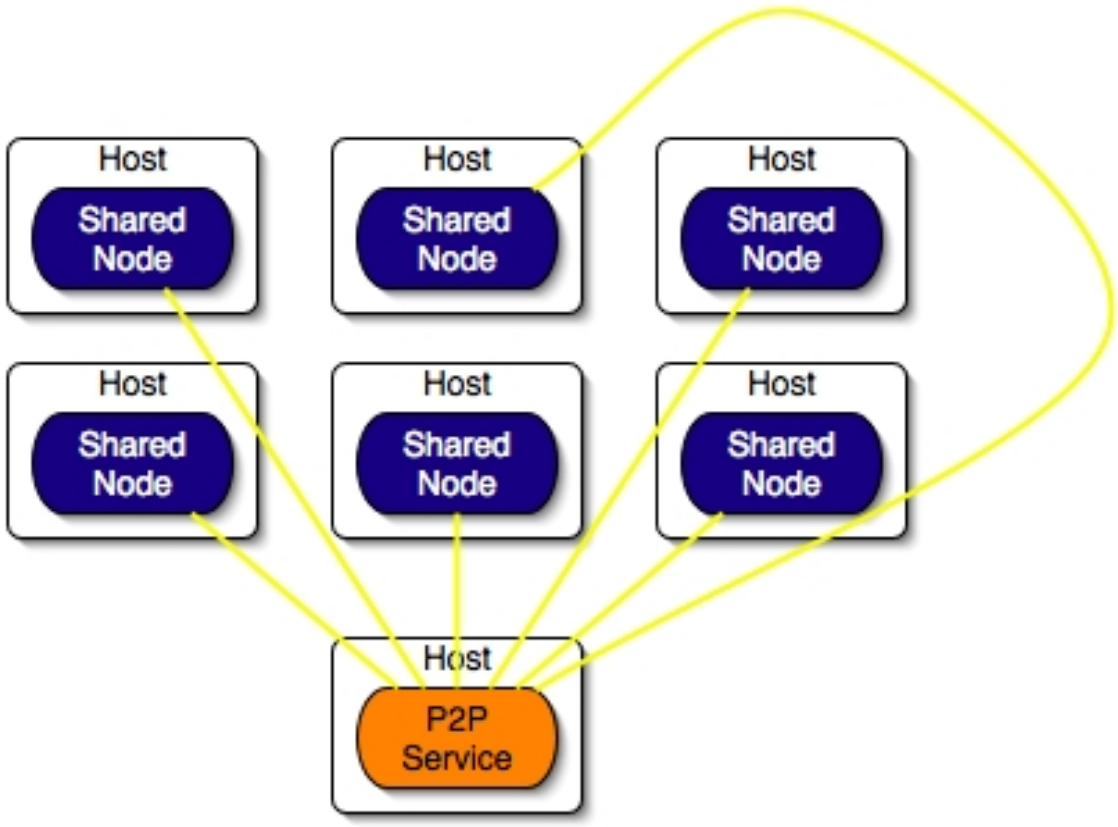
A short preview of a XML descriptor:

```
<infrastructure>
 <services>
  <serviceDefinition id="p2pservice">
   <P2PService nodesAsked="2" acq="rmi"
         port="2410" NOA="10" TTU="60000"
         TTL="10">
    <peerSet>
     <peer>rmi://localhost:3000</peer>
    </peerSet>
   </P2PService>
  </serviceDefinition>
 </services>
```

```
</infrastructure>
```

A complete example of file is available here [p2p_files/sample_p2p.xml].

The next figure shows a P2P Service started with a XML deployment descriptor (xml_path attribute). Six nodes are shared on different hosts:



A P2P Service which is sharing nodes deployed by a descriptor.

For more information about ProActive XML Deployment Descriptor see this page [http://www-sop.inria.fr/oasis/ProActive/doc/api/org/objectweb/proactive/doc-files/Descriptor.xml].

21.4.3. The P2P Infrastructure API Usage Example

The next little sample of code explains how, from an application, you can start a P2P Service and get nodes:

```
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.ProActiveException;
import org.objectweb.proactive.core.mop.ClassNotReifiableException;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.core.node.NodeFactory;
import org.objectweb.proactive.core.runtime.ProActiveRuntime;
import org.objectweb.proactive.core.runtime.RuntimeFactory;
import org.objectweb.proactive.p2p.core.service.P2PService;
import org.objectweb.proactive.p2p.core.service.StartP2PService;
import org.objectweb.proactive.p2p.core.service.node.P2PNodeLookup;
...
```

```
// This constructor uses a file with address of peers
// See the Javadoc to choose different parameters
StartP2PService startServiceP2P = new StartP2PService(p2pFile)
// Start the P2P Service on the local host
startServiceP2P.start();
// Get the reference on the P2P Service
P2PService serviceP2P = startServiceP2P.getP2PService();
// By the application's P2P Service ask to the P2P infrastructure
// for getting nodes.
P2PNodeLookup p2pNodeLookup = p2pService.getNodes(nNodes,
 virtualNodeName, JobID);
// You can migrate the P2P node lookup from the p2p service
// to an another node:
p2pNodeLookup.moveTo("//localhost/localNode");
// Use method from p2pNodeLookup to get nodes
// such as
while (! p2pNodeLookup.allArrived()) {
 Vector arrivedNodes = p2pNodeLookup.getAndRemoveNodes();
 // Do something with nodes
 ...
}
// Your application
...
// End of your program
// Free shared nodes
p2pNodeLookup.killAllNodes();
```

21.5. Future Work

- Plug technical services, such as Fault-tolerance schemes or Load Balancing, for each application at the deployment time.

Chapter 22. ProActive Security Mechanism

In order to use the Proactive Security features, you have to install**the Java(TM) Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files** available at Sun's website [http://www.java.sun.com]. Extract the file and copy jar files to your <jre_home>/lib/security.

22.1. Overview

Usually, applications and security are developed for a specific use. We propose here a security framework that allows to dynamically deploy applications and to configure security according to this deployment.

ProActive security mechanism provides a set of security features from basic ones like communications authentication, integrity, confidentiality to more high-level features including migration security mechanism, hierarchical security policies, dynamically negotiated policies. All theses features are expressed inside the meta-level of the middleware and used transparently by applications.

It is possible to attach security policies to Runtimes, Virtual Nodes, Nodes and Active Objects. Policies are expressed inside an XML descriptor.

22.2. Security Architecture

22.2.1. Base model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object; all other objects inside the active object are called *passive objects* and cannot be referenced directly from objects which are outside this active object (see Figure 8 [security_images/node15.xml#graphe]); the absence of sharing is important with respect to security.

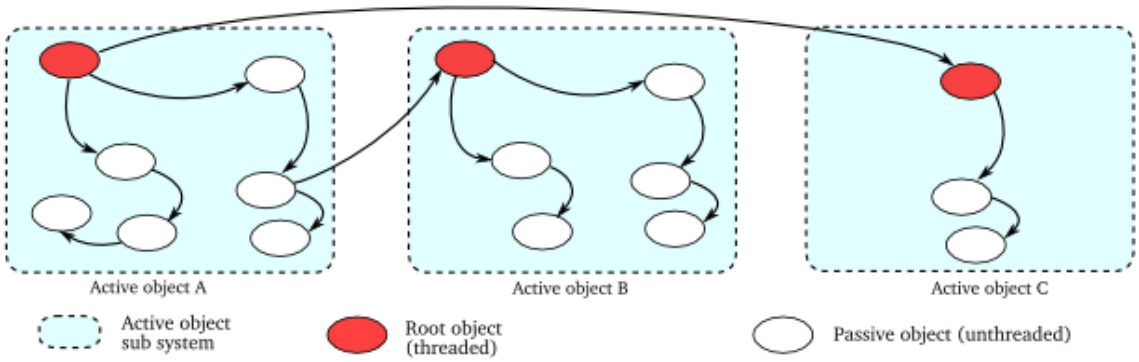


Figure 8: A typical object graph with active objects

The security is based on Public Key Infrastructure. Each entity owns a certificate and an private key generated from the certificate of a user.

Certificates are generated automatically by the security mechanism. The validity of a certificate is checked by validating its certificate chain. As shown in figure 2, before validating the certificate of an active object, application certificate and user certificate will be checked. If a valid path is found so object certificate is validated.

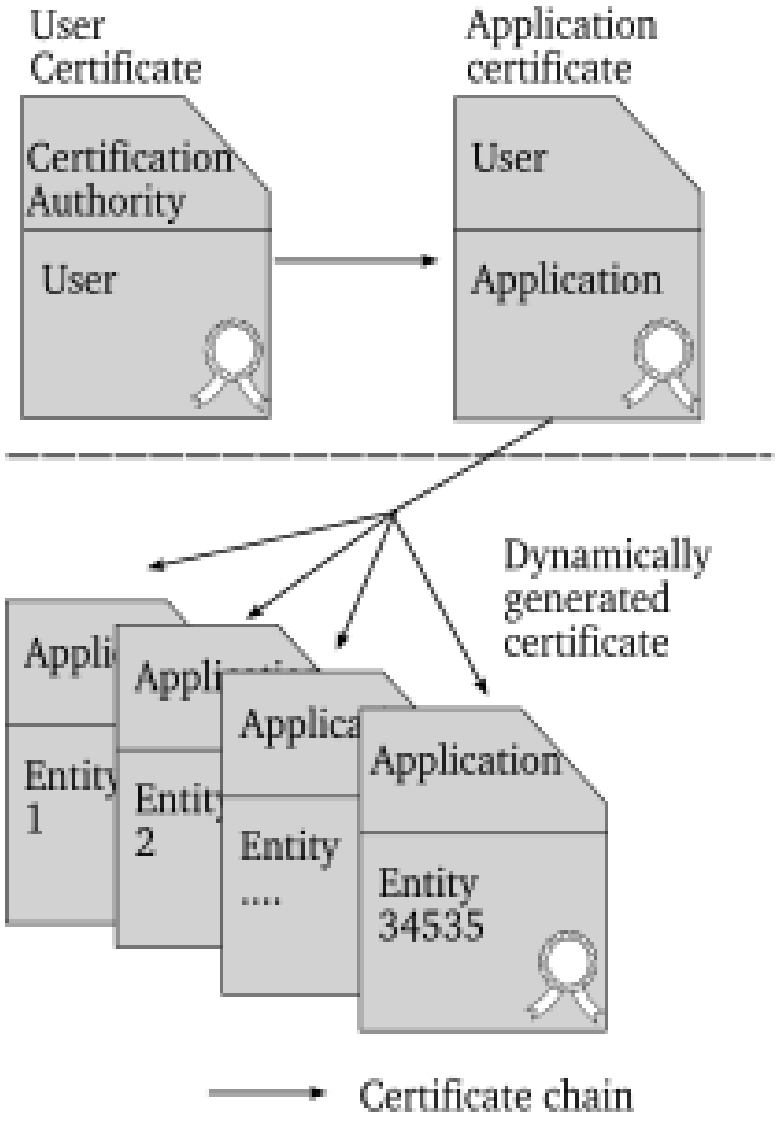


Figure 2 : Certificate chain

22.2.2. Security is expressed at different level according to who wants to set policy :

- Administrators set policy at domain level. It contains general security rules.
- Resource provider set policy for resource. People who have access to a cluster and wants to offer cpu time under some restrictions. The runtime loads its policy file at launch time.
- Application level policy is set when an application is deployed through an XML descriptor.

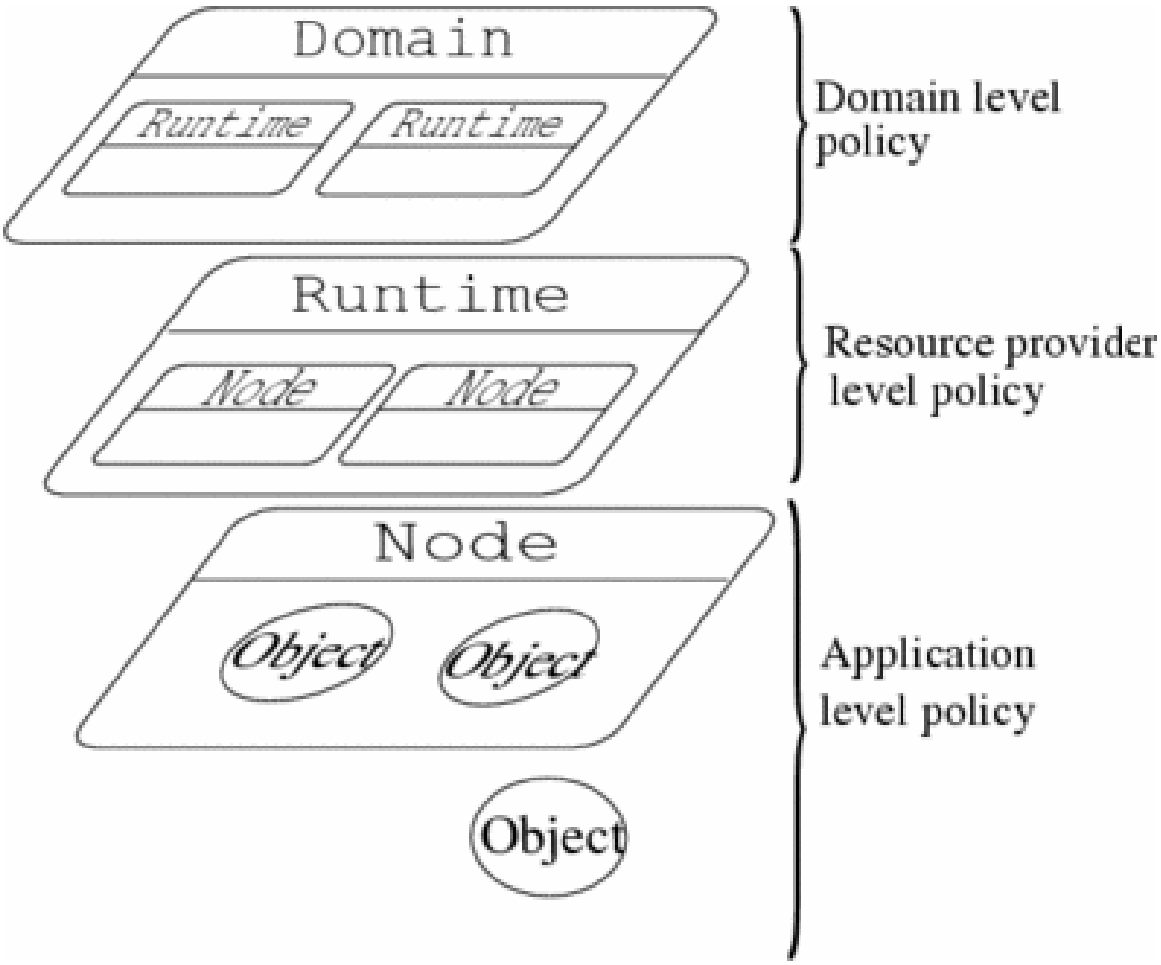


Figure 1 : Hierarchical security

When an interaction is happening, all participating entities' security policy are verified and combined to find the policy to enforce to the interaction.

22.3. Detailed Security Architecture

22.3.1. Virtual Nodes and Nodes

The security architecture relies on two related abstractions for deploying Grid applications: *Node* and *Virtual Node*. A node gathers several objects in a logical entity. It provides an abstraction for the physical location of a set of activities. Objects are bound to a node at creation or after migration. In order to have a flexible deployment (eliminating from the source code machine names, creation protocols), the system relies on *Virtual Nodes* (VNs). A VN is identified as a name (a simple string), used in a program source, defined and configured in an descriptor. The user can attach policy to these virtual nodes. Virtual Nodes are used within application code to structure it. By example, an object which will be used as a server will be set inside a virtual node named Server_VN, client objects will be set inside Client_VN. The user expresses policy between server and client object inside a descriptor file. The correspondence between Virtual Nodes and Nodes, the mapping, is done at application starting time.

22.3.2. Hierarchical Security Entities

Grid programming is about deploying processes (activities) on various machines. In the end, the security policy that must be ensured for those processes depends upon many factors: first of all, the application policy that is needed, but also the machine locations, the security policies of their administrative domain, and the network being used to reach those machines.

Previous section defined the notions of *Virtual Nodes*, and *Nodes*. Virtual Nodes are application abstractions, and nodes are only a run-time entity resulting from the deployment: a mapping of Virtual Nodes to processes and hosts. A first decisive feature allows to define application-level security on those application-level abstractions:

Definition 1. *Virtual Node Security*
Security policies can be defined at the level of Virtual Nodes. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to JVMs, and Hosts.

As such, virtual nodes are the support for intrinsic application level security. If, at design time, it appears that a process always requires a specific level of security (e.g. authenticated and encrypted communications at all time), then that process should be attached to a virtual node on which those security features are imposed. It is the designer responsibility to structure his/her application or components into virtual node abstractions compatible with the required security. Whatever deployment occurs, those security features will be maintained. We expect this usage to be rather occasional, for instance in very sensitive applications where even an intranet deployment calls for encrypted communications.

The second decisive feature deals with a major Grid aspect: deployment-specific security. The issue is actually twofold:

1. allowing organizations (security domains) to specify general security policies,
2. allowing application security to be specifically adapted to a given deployment environment.

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure; they are organized in a hierarchical manner. They are the logical concept allowing to express security policies in a hierarchical way.

Definition 2. *Declarative Domain Security*
Fine grain and declarative security policies can be defined at the level of Domains*. A Security Domain is a domain to which a certificate and a set of rules are associated.*

This principle allows to deal with the two issues mentioned above:

(1) the administrator of a domain can define specific policy rules that must be obeyed by the applications running within the domain. However, a general rule expressed inside a domain may prevent the deployment of a specific application. To solve this issue, a policy rule can allow a well-defined entity to weaken it. As we are in a hierarchical organization, allowing an entity to weaken a rule means allowing all entities included to weaken the rule. The entity can be identified by its certificate;

(2) a Grid user can, at the time he runs an application, specify additional security based on the domains being deployed onto.

The Grid user can specify additional rules directly in his deployment descriptor for the domains he deploys onto. Note that those domains are actually dynamic as they can be obtained through external allocators, or even Web Services in an OGSA infrastructure [5 [security_images/node18.xml#foster98security]]. Joker rules might be important in that case to cover all cases, and to provide a conservative security strategy for un-forecasted deployments.

Finally, as active objects are active and mobile entities, there is a need to specify security at the level of such entities.

Definition 3. *Active Object Security*
Security policies can be defined at the level of Active Object. Upon migration of an activity, the security policy attached to that object follows.

In open applications, e.g. several principals interacting in a collaborative Grid application, a JVM (a process) launched by a given principal can actually host an activity executing under another principal. The principle above allows to keep specific security privileges in such case. Moreover, it can also serve as a basis to offer, in a secure manner, hosting environments for mobile agents.

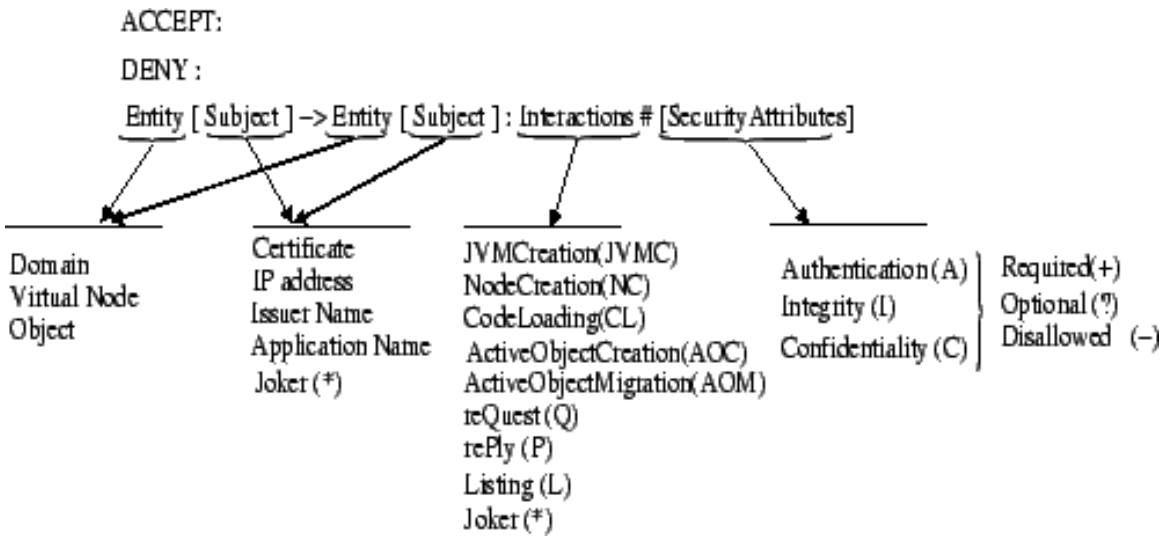


Figure 5: Syntax and attributes for policy rules

22.3.3. Resource provider security features

Prior to start application on a Grid, a user needs to acquire some resources (CPU time, disk storage, bandwidth) from the Grid. A *Resource provider* is an individual, a research institute, an organization who wants to offer some resources under a certain security policy to a restricted set of peoples. According to our definition, resource provider will set up one or more runtime where clients will be able do perform computation. Each runtime is set with its own policy. Theses runtimes could be worldwide distributed.

22.3.4. Interactions, Security Attributes

Security policies are able to control all the *interactions* that can occur when deploying and executing a multi-principals Grid application. With this goal in mind, interactions span over the creation of processes, to the monitoring of activities (Objects) within processes, including of course the communications. Here is a brief description of those interactions:

- RuntimeCreation (RC): creation of a new Runtime process
- NodeCreation (NC): creation of a new Node within a Runtime (as the result of Virtual Node mapping)
- CodeLoading (CL): loading of bytecode within a Node, used in presence of object migration.
- ObjectCreation (OC): creation of a new activity (active object) within a Node
- ObjectMigration (OM): migration of an existing activity object to a Node
- Request (Q), Reply (P): communications, method calls and replies to method calls

- Listing (L): list the content of an entity; for Domain/Node provides the list of Node/Objects, for an Object allows to monitor its activity.

For instance, a domain is able to specify that it accepts downloading of code from a given set of domains, provided the transfers are authenticated and guaranteed not to be tampered with. As a policy might leave open the integrity of communications, and also because not allowing confidentiality can be a domain (or even a country) policy, those 3 security attributes can be specified in 3 modes: Required (+), Optional (**?**), Disallowed (-)

For a given interaction, a tuple [+A,?I,-C] means that authentication is required, integrity is accepted but not required, and confidentiality is not allowed.

As a Grid operates in decentralized mode, without a central administrator controlling the correctness of all security policies, these policies must be *combined*, *checked*, and *negotiated* dynamically. The next two sections present that aspect.

22.3.5. Combining Policies

As the proposed infrastructure takes into account different actors of the Grid (domain administrator, Grid user), even for a single-principal single-domain application, there are potentially several security policies activated. This section deals with the combination of those policies to obtain the final security tuples of a single entity. An important principle being that a sub-domain cannot weaken a super-domain's rule.

During execution, each activity (Active Object) is always included in a *Node* (due to the Virtual Node mapping) and at least in one *Domain*, the one used to launch a JVM (

$$D_0$$

). Figure 6 [security_images/node11.xml#hierarchicalsecurityrule] hierarchically represents the security rules that can be activated at execution: from the top, hierarchical domains (

$$D_n$$

to

$$D_0$$

), the virtual node policy (VN), and the Object (O) policy. Of course, such policies can be inconsistent, and there must be clear principles to combine the various sets of rules.

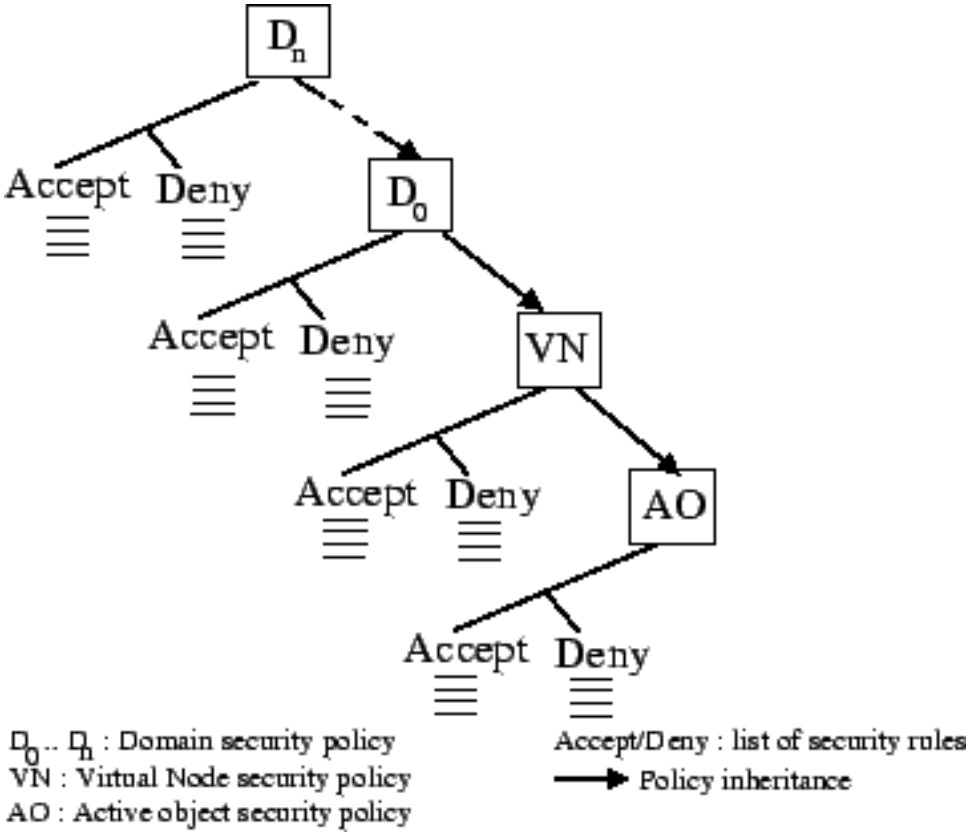


Figure 6: Hierarchical Security Levels

There are three main principles: (1) choosing the *most specific rules* within a given domain (as a single Grid actor is responsible for it), (2) an interaction is valid only if all levels accept it (absence of weakening of authorizations), (3) the security attributes retained are the most constrained based on a partial order (absence of weakening of security).

```
 Domain[*] -> Domain[*] : Q,P : [+A,+I,+C]
Domain[CardPlus] -> Domain[CardPlus] : Q,P : [+A,?I,?C]
```

within the CardPlus domain, the second rule will be chosen (integrity and confidentiality will be optional). Of course, comparison of rules is only a partial order, and several incompatible most specific rules can exist within a single level (e.g. both ACCEPT and DENY most specific rules for the same interaction, or both +A and -A).

Between levels, an incompatibility can also occur, especially if a sub-level attempts to weaken the policy on a given interaction (e.g. a domain prohibits confidentiality [-C] while a sub-domain or the Virtual Node requires it [+C], a domain D_i prohibits loading of code while D_j (j <= i) authorizes it). In all incompatible cases, the interaction is not authorized and an error is reported.

22.3.6. Dynamic Policy Negotiation

During execution, entities interact by pair with each other. Each entity, for each interaction (JVM creation, communication, migration, ...), will want to apply a security policy based on the resolution presented in the previous section. Before starting an interaction, a *negotiation* occurs between the two entities involved. Figure 7 [security_images/node12.xml#sac] shows the result of such negotiation. For example, if for a given interaction, entity A's policy is [+A,?I,?C], and B's policy is [+A,?I,-C], the negotiated policy will be [A,?I,-C]. If, for a result rule, one of the communication attributes is optional, the attribute is not activated.

Besides the interactions not being accepted by an entity, two other cases lead to an error: when an attribute is required by one, and disallowed by the other. In such cases, the interaction is not authorized and an error is reported. If a valid security policy is found between two entities, the interaction can occur. In the case that the agreed security policy includes confidentiality, the two entities negotiate a session key.

		Entity A		
		Required (+)	Optional (?)	Disallowed (-)
Entity B	Required (+)	+	+	Error
Optional (?)	+	?	-	
Disallowed (-)	Error	-	-	

Figure 7: Result of security negotiations

22.3.7. Migration and Negotiation

In large scale Grid applications, migration of activities is an important issue. The migration of Active Objects must not weaken the security policy being applied.

When an active object migrates to a new location, three cases may happen :

- the object migrates to a node belonging to the same virtual node and included inside the same domain. In this case, all already negotiated sessions remain valid.
- the object migrates to a known node (created during the deployment step) but which belongs to another virtual node. In this case, all already negotiated sessions can be invalid. This kind of migration imposes re-establishing the object policy, and upon a change, re-negotiating with interacting entities.
- The object migrates to an unknown node (not known at the deployment step). In this case, the object migrates with a copy of the application security policy. When a secured interaction will take place, the security system retrieves not only the object's application policy but also policies rules attached to the node on which the object is to compute the policy.

22.4. Activating security mechanism

Within descriptor, the tag <security> is used to specify the policy for the deployed application. It will be the policy for all Nodes and active that will be created.

the descriptor is:

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:<ProActiveDescriptor
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
 3: <security file="../../descriptors/applicationPolicy.xml"></security>
 4: <componentDefinition>
 5: <virtualNodesDefinition>
 6: <virtualNode name="
Locale
" property="unique"/>
 7: <virtualNode name="
vm1
" property="unique"/>
 8: <virtualNode name="
vm2
" property="unique"/>
 9: </virtualNodesDefinition>
 10: </componentDefinition>
 11: ....
 50:<infrastructure>
 51: <processes>
 52: <processDefinition id="linuxJVM">
 53: <jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess">
 54: <classpath>
 ....
 74: </classpath>
 75: <jvmParameters>
 <parameter
value="-Dproactive.runtime.security=/.../descriptors/jvm1-sec.xml" \
/>
 82: </jvmParameters>
 83: </jvmProcess>
 84: </processDefinition>
 ....
```

Inside the policy file, you can express policy between entities (domain, runtime, node, active object).

The entity tag can be used to :

- express policies on entities described inside the descriptor (lines 13, 15)
- express policies on existing entities by specifying theirs certificates (line 32).

22.4.1. Construction of an XML policy :

A policy file must begin with :

```
1:<?xml version="1.0" encoding="UTF-8"?>
2:<Policy>
```

next, application specific informations are given.

```
3: <ApplicationName>Garden</ApplicationName>
```

<ApplicationName> sets the application name. This allows to identify easily which application an entity belongs to.

```
4: <Certificate>/.../appli.cert</Certificate>
5: <PrivateKey>/.../appli.key</PrivateKey>
```

<Certificate> is the X509 certificate of the application, generated from a user certificate, and

<PrivateKey> the private key associated to the certificate.

```
 6: <CertificationAuthority>
7: <Certificate>/.../ca.cert</Certificate>
8: </CertificationAuthority>
```

<CertificationAuthority> contains all trusted certificate authority. Each <Certificate> contains a certification authority certificate.

```
 10: <Rules>
```

Then we can define policy rules. All rules are located within the <Rules>

A <Rule> is constructed according the following syntax :

```
11: <Rule>
```

<From> tag contains all entities from which the interaction is made. It is possible to specify many entities in order to match a specific fine-grained policy.

```
12: <From>
13: <Entity type="VN" name="vm2"/>
14: </From>
```

<Entity> is used to define an entity. the "type" parameter can be "VN", "certificate".

- "VN" (Virtual Node) referrers to virtual nodes defined inside the deployment descriptor.
- "DefaultVirtualNode" is a special tag. It allows to specify a default policy.
- "certificate" supposes that a path to a file is set inside the "name" parameters.

<To> tag contains all entities onto the interaction is made. As <From> tag, many entities can be specified.

```
15: <To>
16: <Entity type="VN" name="Locale"/>
17: </To>
```

The <Communication> tag defines security policies to apply to requests and replies.

```
18: <Communication>
```

<Request> sets the policy associated a request. the "value" parameter can be :

- "authorized" means a request is authorized.
- "denied" means a request is denied.

Each <Attribute> (authentication,integrity, confidentiality) can be required, optional or denied.

```
 19: <Request value="authorized">
 20:  <Attributes authentication="required" integrity="optional"↩
confidentiali\
ty="optional"/>
 21: </Request>
```

<Reply> tag has the same parameters that <Request>

```
22:<Reply value="authorized">
23: <Attributes authentication="required" integrity="required"↩
confidentiality="\
required"/>
24:</Reply>
25:</Communication>
```

<Migration> allows or not migration from <from> entities to <To> entities. Values can be "denied" or "authorized".

```
26: <Migration>denied</Migration>
```

<OACreation> allows or not creation of active objects by <From> entities onto <To> entities.

Values can be "denied" or "authorized".

```
 27: <OACreation>denied</OACreation>
```

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:<Policy>
 3: <ApplicationName>Garden</ApplicationName>
 4: <Certificate>/net/home/acontes/certif/appli.cert</Certificate>
 5: <PrivateKey>/net/home/acontes/certif/appli.key</PrivateKey>
 6: <CertificationAuthority>
 7: <Certificate></Certificate>
 8: </CertificationAuthority>

 10: <Rules>
 11: <Rule>
 12: <From>
 13: <Entity type="VN" name="vm2"/>
 14: </From>
 15: <To>
 16: <Entity type="VN" name="Locale"/>
 17: </To>
 18: <Communication>
 19: <Request value="authorized">
 20: <Attributes authentication="required" integrity="required"↩
confidentiality="\
required"/>
 21: </Request>
 22: <Reply value="authorized">
 23: <Attributes authentication="required" integrity="required"↩
confidentiality="\
required"/>
 24: </Reply>
 25: </Communication>
 26: <Migration>denied</Migration>
 27: <OACreation>denied</OACreation>
 28:
 29: </Rule>
 30: <Rule>
 31: <From>
 32: <Entity type="certificate" name="certificateRuntime1.cert"/>
 33: </From>
 34: <To>
 35: <Entity type="VN" name="Locale"/>
 36: </To>
 37: <Communication>
 38: <Request value="authorized">
 39: <Attributes authentication="required" integrity="required"↩
confidentiality="\
required"/>
 40: </Request>
 41: <Reply value="authorized">
 42: <Attributes authentication="required" integrity="required"↩
confidentiality="\
```

```
required"/>
 43: </Reply>
 44: </Communication>
 45: <Migration>denied</Migration>
 46: <OACreation>denied</OACreation>
 47:
 48: </Rule>
 ...
 90: <Rule>
 91: <From>
 92: <Entity type="DefaultVirtualNode" name="*"/>
 93: </From>
 94: <To>
 95: <Entity type="DefaultVirtualNode" name="*"/>
 96: </To>
 97: <Communication>
 98: <Request value="denied">
 99: <Attributes authentication="optional" integrity="optional"↩
confidentiality="\
optional"/>
 100: </Request>
 101: <Reply value="denied">
 102: <Attributes authentication="optional" integrity="optional"↩
confidentiality\
="optional"/>
 103:
 104: </Reply>
 105: </Communication>
 106: <Migration>denied</Migration>
 107: <OACreation>authorized</OACreation>
 108:
 109: </Rule>
 110:
 111: </Rules>
 112:</Policy>
```

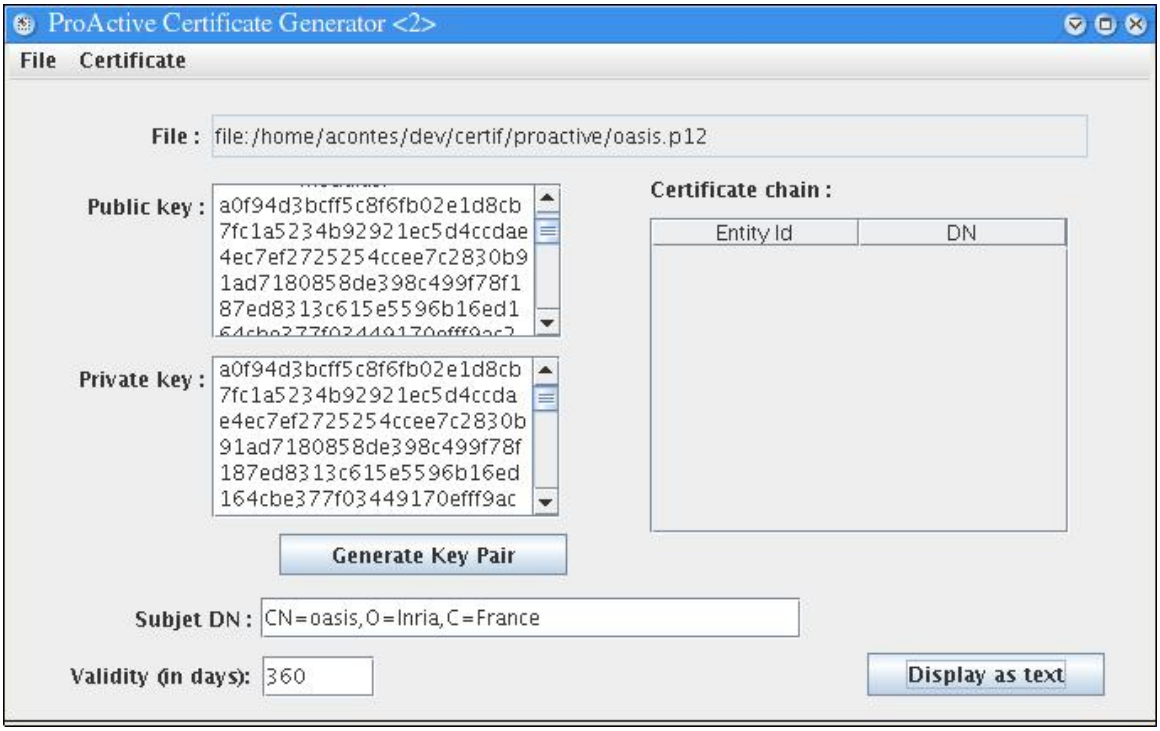
Note that the JVM that reads the deployment descriptor should be started with security policy. In order to start a secure JVM, you need to use the property proactive.runtime.security and give a path a security file descriptor.

Here an example : java -Dproactive.runtime.security=jvmlocal.xml TestSecureDeployment secureDeployment.xml

22.5. How to quickly generate certificate ?

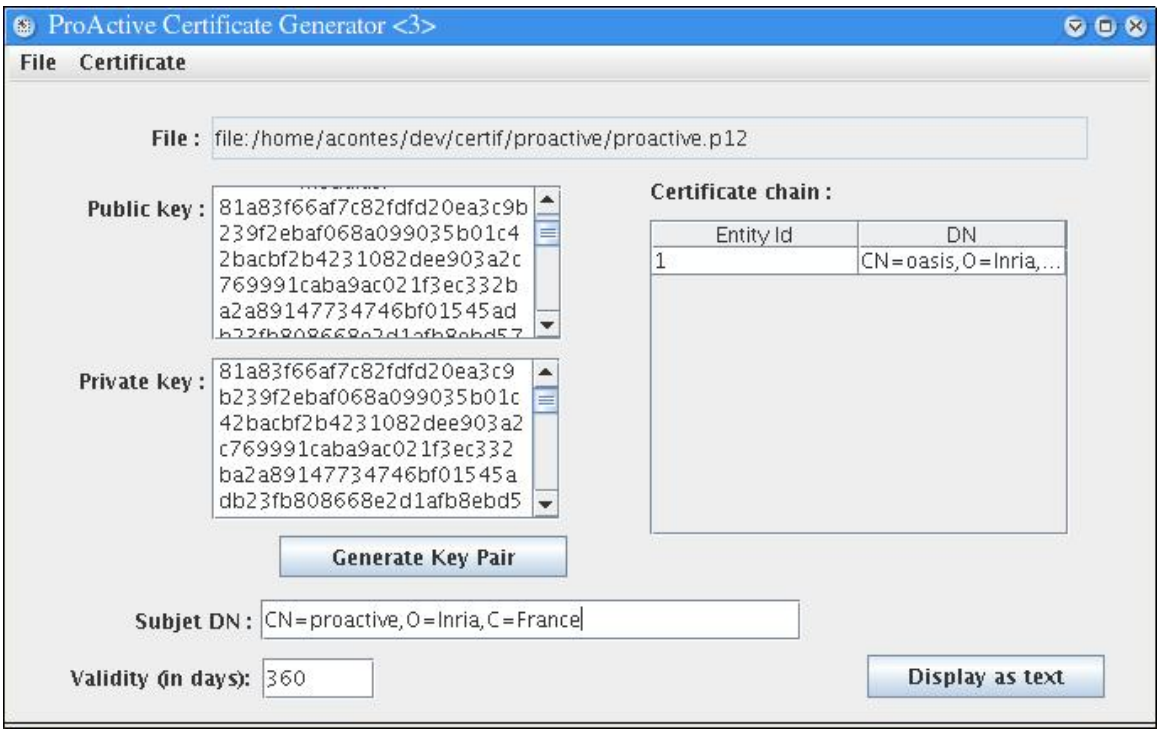
A GUI has been created to facilitate certificate generation.

The first screenshot presents a root certificate. Notice that the certificate chain table is empty.



The second screenshot presents a certificate generated from the previous one using menu entry "Certificate -> generate a sub-certificate".

Notice that the certification table contains one entry and Distinguished Name of the Entity ID 1 is the same as the subject DN of the certificate



Using this GUI, a user is able to generate a certificate and if needed a certificate chain.

Certificates are saved under a PKCS12 format (extension .p12). This format is natively supported by the ProActive Security mechanism.

To learn more, see the JavaDoc [../../../../index.xml].

Chapter 23. Exporting Active Objects and components as web services

23.1. Overview

This feature allows the call and monitoring of active objects and ProActive components from any client written in any foreign language.

Indeed, applications written in C#, for example, cannot communicate with ProActive applications. We choose the web services technology that enable interoperability because they are based on XML and HTTP. Thus, any active object or component can be accessible from any enabled web service language.

23.2. Principles

A **web service** is a software entity, providing one or several functionnalities, that can be exposed, discovered and accessed over the network. Moreover, web services technology allows heterogenous applications to communicate and exchange data in a remotely way. In our case, the usefull elements, of web services are :

- **The SOAP Message**

  The SOAP message is used to exchange XML based data over the internet. It can be sent via HTTP and provides a serialization format for communicating over a network.

- **The HTTP Server**

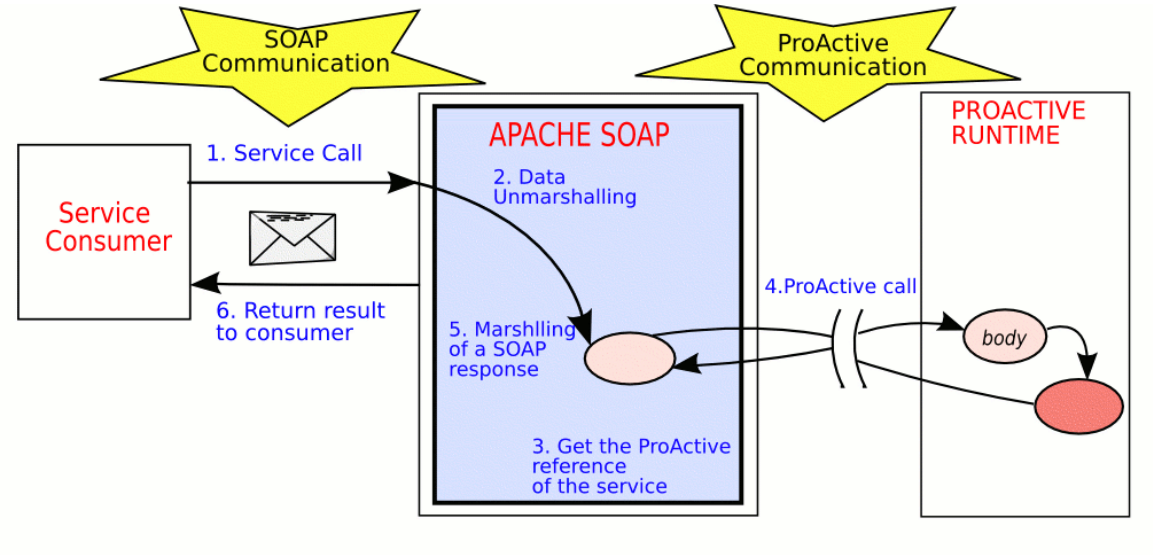
  HTTP is the standard web protocol generally used over the 80 port. In order to receive SOAP messages you need to install an HTTP server that will be responsible of the data transfer. This server is not sufficient to treat a SOAP request.

- **The SOAP Engine**

  A SOAP Engine is the mechanism responsible of making transparent the unmarshalling of the request and the marshalling of the response. Thus, the service developer doesn't have to worry with SOAP. In our case, we use Apache SOAP which is installed on a Jakarta Tomcat web server. Moreover, Apache SOAP contains a web based administration tool that permit to list, deploy and undeploy services.

- **The client**

  Client's role is to consume a web service. It is the producer of the SOAP message. The client developer doesn't have to worry about how the service is implemented.



This figure shows the steps when a active object is called via SOAP.

23.3. Pre-requisite : Installing the Web Server and the SOAP engine

First of all, you need to install the Jakarta Tomcat web server here and install it. You can find some documentation about it here [http://apache.crihan.fr/dist/jakarta/tomcat-4/v4.1.31/bin/] .

You don't really have to do a lot of installation. Just uncompress the archive.

To start and stop the server, launch the start and the shutdown scripts in the bin directory.

We also use a SOAP engine which is the Apache SOAP engine, available here [http://www.apache.org/dyn/closer.cgi/ws/soap/] . This SOAP engine will be responsible of locating and calling the service.

To install Apache SOAP refer to the server-side instructions. [http://ws.apache.org/soap/docs/install/index.xml]

The SOAP Engine is now installed ! You can verify, after starting the server that you access to the welcome page of Apache SOAP at : http://localhost:8080/soap/index.html [http://localhost:8080/soap/index.xml].

Now we have to install ProActive into this SOAP engine. For that, follow these steps :

- Copy the ProActive.jar file into the $APACHE-SOAP/WEB-INF/lib/
- Replace the $TOMCAT/webapps/soap/WEB-INF/web.xml by this one [web.xml]

23.4. Steps to expose an active object or a component as a web services

The steps for exporting and using an active object as a web service are the following :

• Write your active object or your component in a classic way; for example:

```
A a = (A)ProActive.newActive("A", new Object [] {});
```

• Once the element is created and activated, deploy it onto a web server by using :

  • For an active object :

```
ProActive.exposeAsWebService(Object o,
                      String url,
                      String urn,
                      String [] methods);
```

  where :

  • **o** is the active object

  • **url** is the url of the web server; typically http://localhost:8080.

  • **urn** is the service name which identify the active object on the server.

  • **methods** a String array containing the methods name you want to make accessible. If this parameter is null, all the public methods will be exposed.

  • For a component :

```
Proactive.exposeComponentAsWebService(Component component,
                                      String url,
                                      String
                                      componentName);
```

  where :

  • **component** is the component whose interfaces will be exposed as web services

  • **url** is the url of the web server; typically http://localhost:8080.

  • **componentName** is the name of the component. Each service available in this way will get a name composed by the component name followed by the interface name : *componentName_interfaceName*

23.5. Undeploy the services

To undeploy an active object as a service, use the ProActive static method :

```
ProActive.unExposeAsWebService ( String urn, String url );
```

where :

- **urn** is the service name
- **url** the url of the server where the service is deployed

To undeploy a component you have to specify the component name and the component(needed to know the interfaces to undeploy) :

```
ProActive.unExposeAsWebService ( String componentName , String url, ↩
Component component);
```

23.6. Accessing the services

Once the active object or the interfaces component are deployed, you can access it via any web service enabled client (such as C#).

First of all, the client will get the WSDL file matching this active object. This WSDL file is the "identity card" of the service. It contains the web service public interfaces and its location. Generally, WSDL files are used to generate a proxy to the service. For example, for a given service, say "compute" ,you can get the WSDL document at http://localhost:8080/servlet/wsdl?id=compute [#].

Now that this client knows what and where to call the service, it will send a SOAP message to the web server, the web server looks into the message and perform the right call then returns the reply into another SOAP message to the client.

23.7. Limitations

Apache Soap supports all defined types in the SOAP 1.1 specification. All Java primitive types are supported but it is not always the case for complex types. For Java Bean Objects, ProActive register them in the Apache SOAP mapping registry, in order to use a specific (de)serializer when such objects are exchanged. All is done automatically, you don't have to matter about the registering of the type mapping. However, if the methods attributes types or return types are Java Beans, you have to copy the beans classes you wrote into the `$APACHE_SOAP_HOME/WEB_INF/classes`.

23.8. A simple example : Hello World

23.8.1. Hello World web service code

Let's start with a simple example, an Hello world active object exposed as a web service :

```
public class HelloWorld implements Serializable {
  public HelloWorld () {}
  public String helloWorld (String name) {
```

```
    return "Hello world !";
  }
  public static void main (String [] args) {

   try {
      HelloWorld hw = (HelloWorld)ProActive.newActive("HelloWorld", new Obj\
ect []{});
      ProActive.exposeAsWebService(hw,
      "helloWorld","http://localhost:8080", new String [] { "helloWorld" });

   } catch (ActiveObjectCreationException e) {
         e.printStackTrace();
   } catch (NodeException e) {
         e.printStackTrace();
   }
  }
}
```

The active object `hw` has been deployed as a web service on the web server located at `"http://localhost:8080"` . The accessible service method is `helloWorld`.

Now that the server-side Web service is deployed, we can create a new client application in Visual Studio .NET.

23.8.2. Access with Visual Studio

In your new Visual Studio Project :

- In the Solution Explorer window, right-click References and click Add Web Reference.
- In the address box enter the WSDL service address, for example : http://localhost:8080/soap/servlet/wsdl?id=helloWorld [#] .When clicking the "add reference" button, this will get the service's WSDL and creates the specific proxy to the service.
- Once the web reference is added, you can use the helloWorld service as an object and perform calls on it :

```
...
localhost.helloWorld hw = new localhost.helloWorld();
string s = hw.helloWorld ();
...
```

23.9. C# interoperability : an example with C3D

23.9.1. Overview

C3D [http://www-sop.inria.fr/oasis/ProActive/apps/c3d.xml] is a Java benchmark application that measures the performance of a 3D raytracer renderer distributed over several Java virtual machines using ProActive. C3D is composed of several parts : the distributed engine (renderers) and the dispatcher that is an active objet. This dispactcher permits users to see the 3D scene and to collaborate. Users can send messages and render command to this dispatcher. This enhancement of C3D is to send commands to the dispatcher from any language. To perform such an enhancement, the Dispatcher object must be exposed as a web service in order to a C# client for example controls it. Only one instruction has been added in the main method :

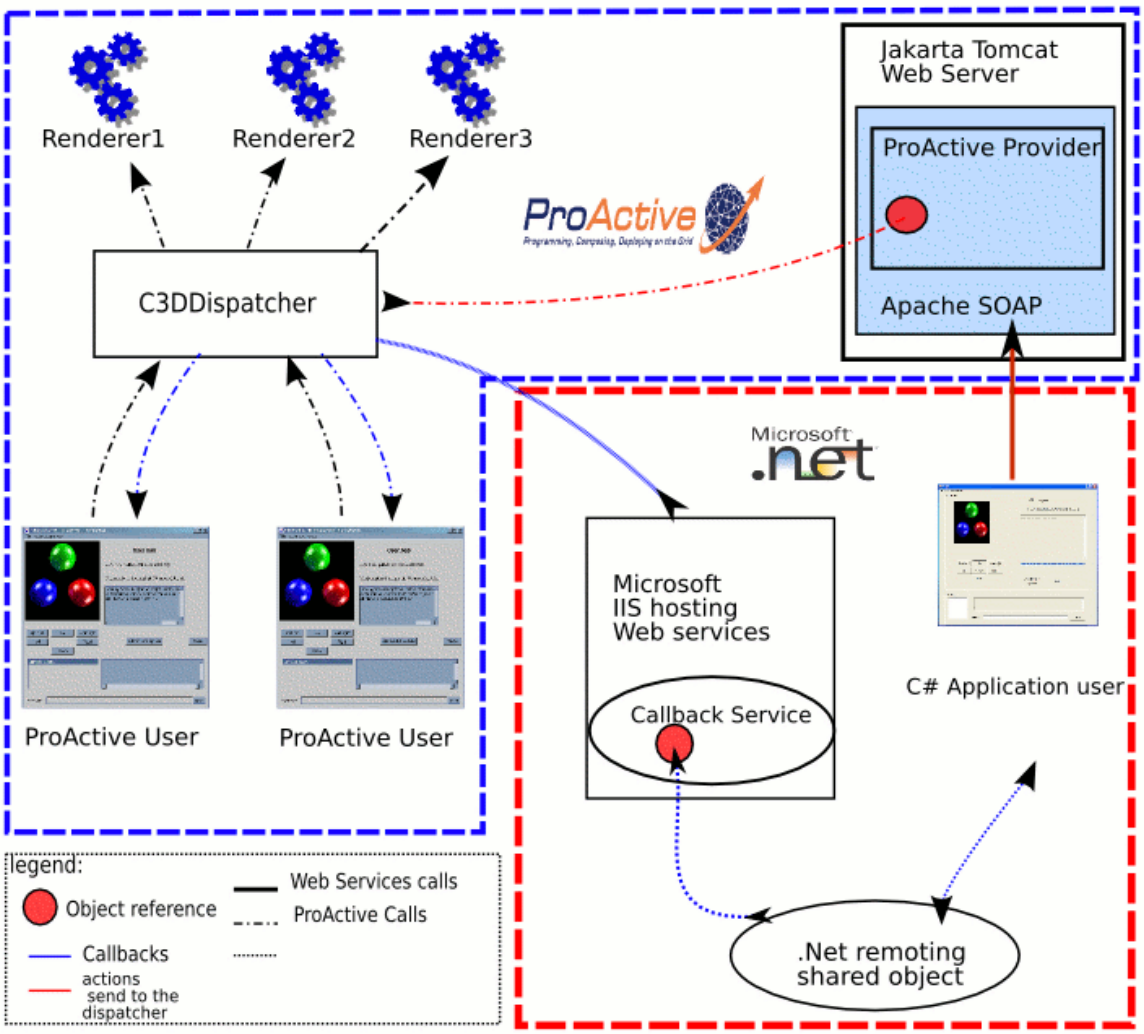
```
ProActive.exposeAsWebService (dispatcher, "C3DDispatcher",
              "http://localhost:8080",  new String [] {
              "rotateRight", "getPicture", "rotateLeft", "rotateUp",
              "rotateDown", "getPixels", "getPixelMax", "waitForImage",
              "spinClock", "spinUnclock", "addRandomSphere", "resetScene",
              "registerWSUser", "unregisterWSUser"
           });
```

Once the dispatcher is deployed as a web service, we have a WSDL url : http://localhost:8080/soap/servlet/id=C3DDispactcher [#] It will be usefull to construct the dispatcher client.

23.9.2. Access with a C# client

First of all, we have to generate the service proxy following the steps described for the hello world access.

All the SOAP calls will be managed by the generated proxy `localhost.C3DDispatcher` .



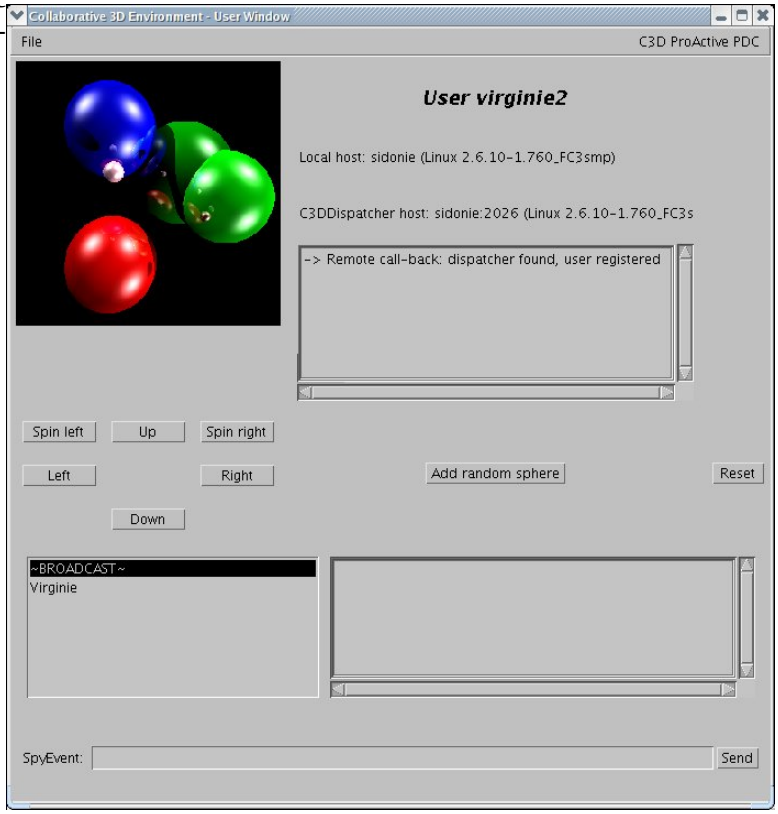
23.9.3. Dispatcher methods calls and callbacks

C# client registers to the C3D dispatcher and then can send commands. C3D is a collaborative application. Indeed, when a client performs a call, all others users must be advised by the dispatcher. Although dispatcher can contact ProActive applications, it cannot communicate with other applications (it cannot initiate the communication). In other words, the dispacher must communicate remotely with an application witten in another language.

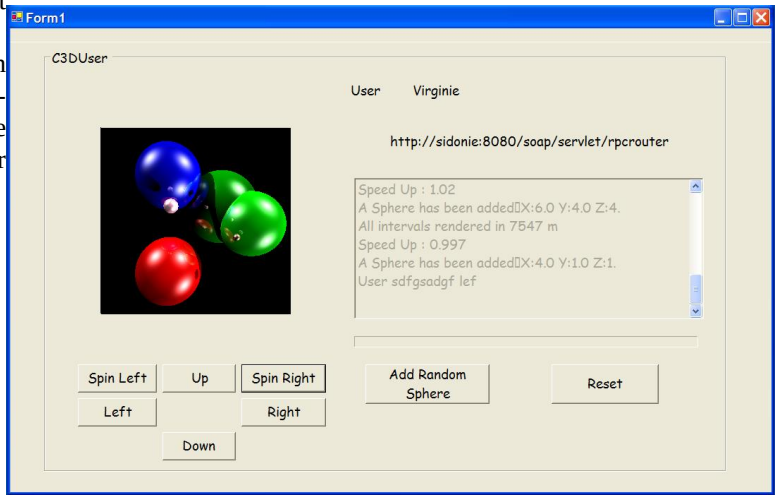
The answer to this problem is to use .Net web service on the C# user machine. Such a web service is waiting for callback requests that come from dispatcher. When receiving a request, the service sends it to the client via a .Net Remoting shared objet. Thus, when the .Net web service receives a callback request, the C# client is updated thanks to propagated events.

Here are screenshots of the user application :

The first screenshot is a classic ProActive application ;



This is the C# application that communicates via SOAP with the same dispatcher than the ProActive user uses.



23.9.4. Download the C# example

You can find here [http://www-sop.inria.fr/oasis/proactive/C3DCSharp.zip] the whole C# Visual Studio .Net project. N.B : In order to run this project, you must install the Microsoft IIS server.

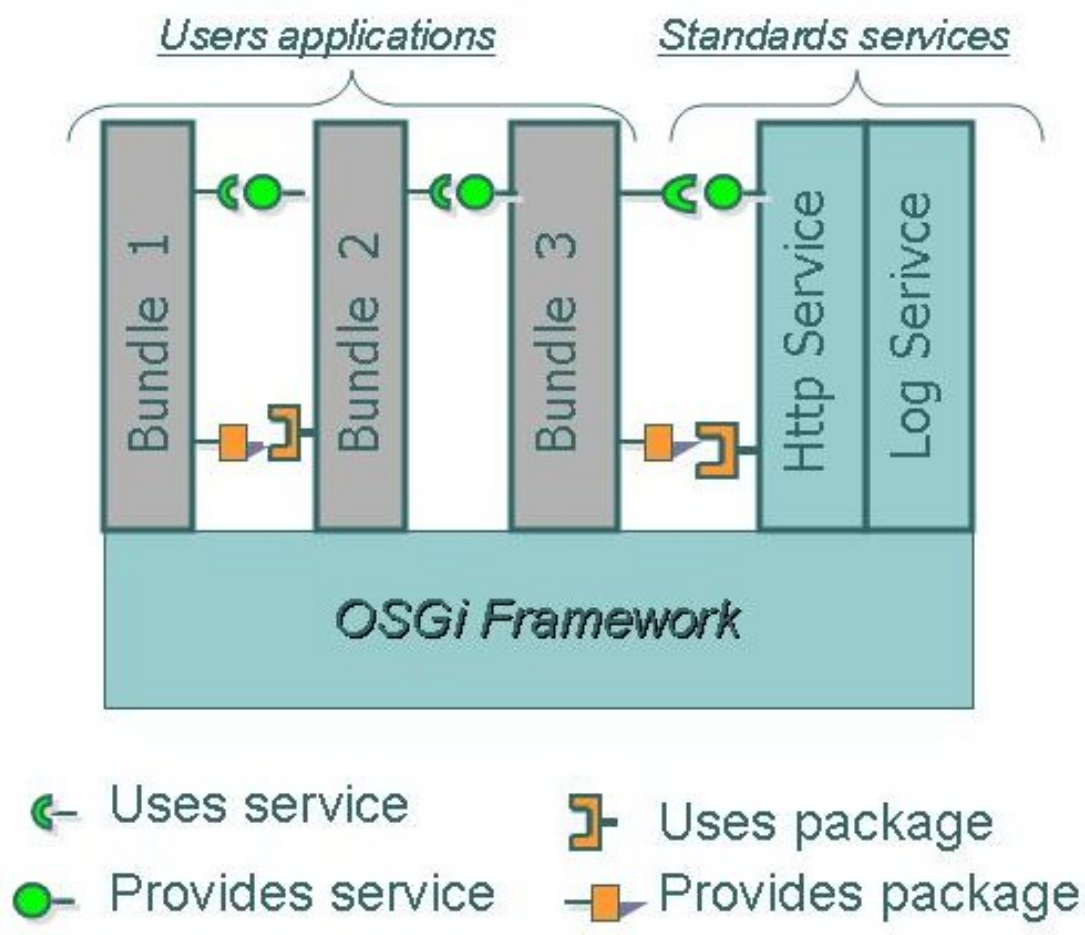
Chapter 24. ProActive on top of OSGi

24.1. Overview of OSGi -- Open Services Gateway initiative

OSGi is a corporation that works on the definition and promotion of open specifications. These specifications are mainly aimed to packaging and delivering services among all kinds of networks.

OSGi Framework

The OSGi specification define **a framework** that allows to a diversity of services to be executed in a service gateway, by this way, many applications can **share a single JVM** .



The OSGi framework entities.

Bundles

In order to be **delivered and deployed** on OSGi, each piece of code is packaged into bundles. Bundles are functionnal entities offering **services and packages**. They can be delivered dynamically to the framework. Concretely a bundle is a Java jar file containing :

- The application classes, including the so called bundle *Activator*
- The *Manifest file* that specifies properties about the application, for instance, which is the bundle Activator, whick packages are required by the application
- Other resources the application could need (images, native libraries, or data files ...) .

Bundles can be plugged dynamically and their so called *lifecycle* can be managed through the framework (start, stop, update, ...).

Manifest file

This important file contains crucial parameters for the bundle file. It specifies which Activator (entry point) the bundle has to use, the bundle classpath, the imported and exported packages, ...

Services

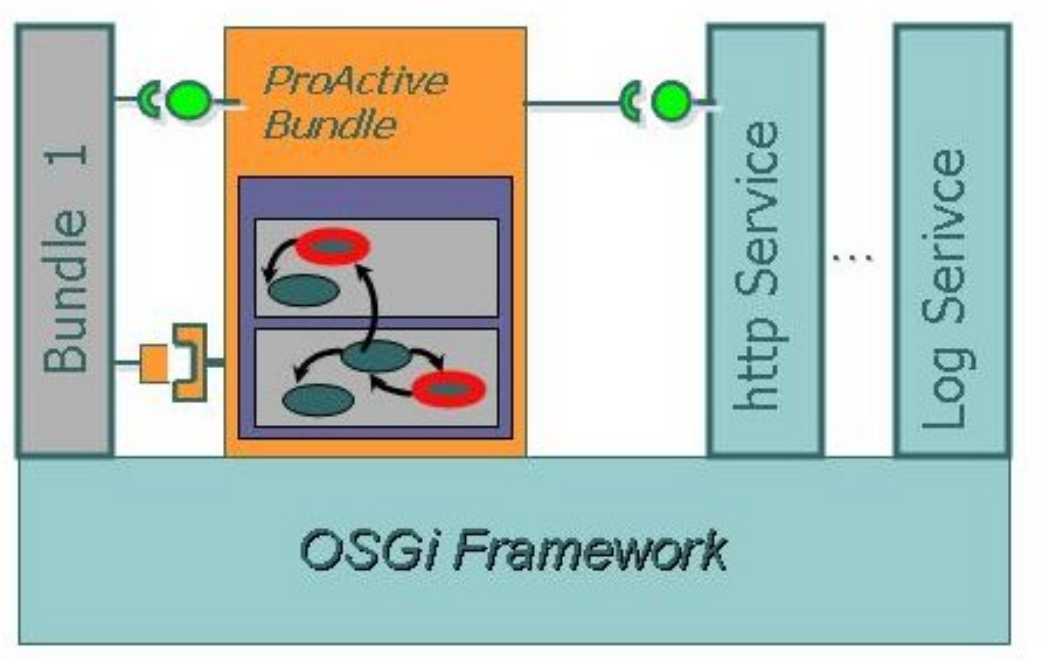
Bundles communicates with each other thanks to **services and packages sharing**. A **service** is an object registered into the framework in order to be used by other applications. The definition of a service is specified in a Java interface. OSGi specify a set of standard services like Http Service, Log Service ...

We currently use the http://oscar.objectweb.org OSCAR objectweb implementation. For more information on OSGi, please visit the http://www.osgi.orgOSGi website .

24.2. ProActive bundle and service

In order to use ProActive on top of OSGi, we have developped the **ProActive Bundle** that contains all classes required to launch a ProActive runtime.

The **ProActive bundle offers a service** , the *ProActiveService* that have almost the same interface that the ProActive static classe. When installed, the ProActive bundle starts a new runtime, and clients that use the ProActive Service will be able to create active object on this runtime.



The Proactive Bundle uses the standard Http Service

The active object will be accessible remotely from any java application, or any other OSGi gateway. The communication can be either *rmi* or *http*; in case of using http, the ProActive bundle requires the installation of the **Http Service** that will handle http communications through a Servlet. We show in the example section how to use the ProActive service.

24.3. Yet another Hello World

The example above is **a simple hello world that uses ProActive Service** . It creates an Hello active Object and register it as a service. We use the hello basic service in the ProActive example. We have to write a **bundle Activator** that will create the active object and register it as a OSGi service.

The HelloActivator has to implements the BundleActivator interface.

```
public class HelloActivator implements BundleActivator {
      ...
}
```

The `start ()` method is the one executed when the bundle starts. When the hello bundle start we need to get the reference on the ProActive service and use it. Once we have the reference, we can create our active object thanks to the `ProActiveService.newActive()` method. Finally, we register our new service in the framework.

```
 public void start(BundleContext context) throws Exception {
      this.context = context;
      /* gets the ProActive Service */
      ServiceReference sr = this.context.getServiceReference(ProActiveSer\
vice.class.getName());
      this.proActiveService = (ProActiveService) this.context.getService(\
sr);
      Hello h = (Hello)this.proActiveService.newActive("org.objectweb.proa\
ctive.examples.hello.Hello",
                                                       \
                    new Object [] {});
      /* Register the service */
      Properties props = new Properties();
      props.put("name", "helloWorld");

      reg = this.context.registerService("org.objectweb.proactive.osgi.Pro\
ActiveService",
                                                 h, props);
      }
```

Now that we created the hello active service, we have to **package the application** into a bundle. First of all, we have to write a **Manifest File** where we specify :

- The name of the bundle : `Hello World ProActive Service`
- The class of the Activator : `org.objectweb.proactive.HelloActivator`

- The packages our application requires : `org.objectweb.proactive....`
- The packages our application exports : org.objectweb.proactive.examples.osgi.hello
- We can specify others informations like author, ...

Here is what the Manifest looks like :

```
Bundle-Name: ProActive Hello World Bundle
Bundle-Description: Bundle containing Hello World ProActive example
Bundle-Vendor: OASIS - INRIA Sophia Antipolis
Bundle-version: 1.0.0
Export-Package: org.objectweb.proactive.examples.hello
DynamicImport-Package: org.objectweb.proactive ...
Bundle-Activator:↩
org.objectweb.proactive.examples.osgi.hello.HelloActivato\
r
```

Installing the ProActive Bundle and the Hello Bundle.

In order to run the example you need to install an OSGi framework. You can download and install one from the OSCAR website [http://oscar.objectweb.org]. Install the required services on the OSCAR framework :

```
--> obr start "Http Service"
```

-
- *Generation of the ProActive Bundle*

  To generate the proActiveBundle you have to run the **build** script with the **`proActiveBundle`** target.

  ```
  > cd $PROACTIVE_DIR/compile
  > ./build proActiveBundle
  ```

  The bundle jar file will be generated in the **`$PROACTIVE_DIR/dist/ProActive/bundle/`** directory. We need now to install and start it into the OSGi Framework :

  ```
  --> start file:///$PROACTIVE_DIR/dist/ProActive/bundle/proActiveBundle.jar
  ```

- This command will install and start the proActive bundle on the gateway. Users can now deploy application that uses the ProActiveService.

- *Generation of the Hello World example bundle*

  To generate the Hello World bundle you have to run the **build** script with the **`helloWorldBundle`** target.

```
> cd $PROACTIVE_DIR/compile
> ./build helloWorldBundle
```

  The bundle jar file will be generated in the **`$PROACTIVE_DIR/dist/ProActive/bundle/`** directory. We need now to install and start it into the OSGi Framework :

```
--> start file:///$PROACTIVE_DIR/dist/ProActive/bundle/helloWorldBundle.jar
```

- The command will install and start the Hello active service. The hello service is now an OSGi service and can be accessed remotely.

24.4. Current and Future works

- We are working on a management application that remotely monitors and manages a large number of OSGi gateways. It uses standard Management API such as JMX (Java Management eXtension). We are writing a ProActive Connector in order to access these JMX enabled gateways and uses Group Communications to handle large scale. Moreover, this application will be graphically written as an Eclipse plugin.
- We plan to deploy remotely active objects and fractal components on OSGi gateways.

Part VI. User Interface and tools

Table of Contents

Chapter 25. IC2D: Interactive Control and Debugging of Distribution

IC2D is a **graphical environment** for remote monitoring and steering of **distributed and grid applications**. IC2D features **graphical visualisation** and **drag and drop migration** of remote objects. As it is being interfaced with **Jini** and **Globus**, it can also serve as a building block for **grid** and **computing portals**. **IC2D** is built on top of **RMI** and **ProActive** that provides asynchronous calls and migration.

The basic features of IC2D are:

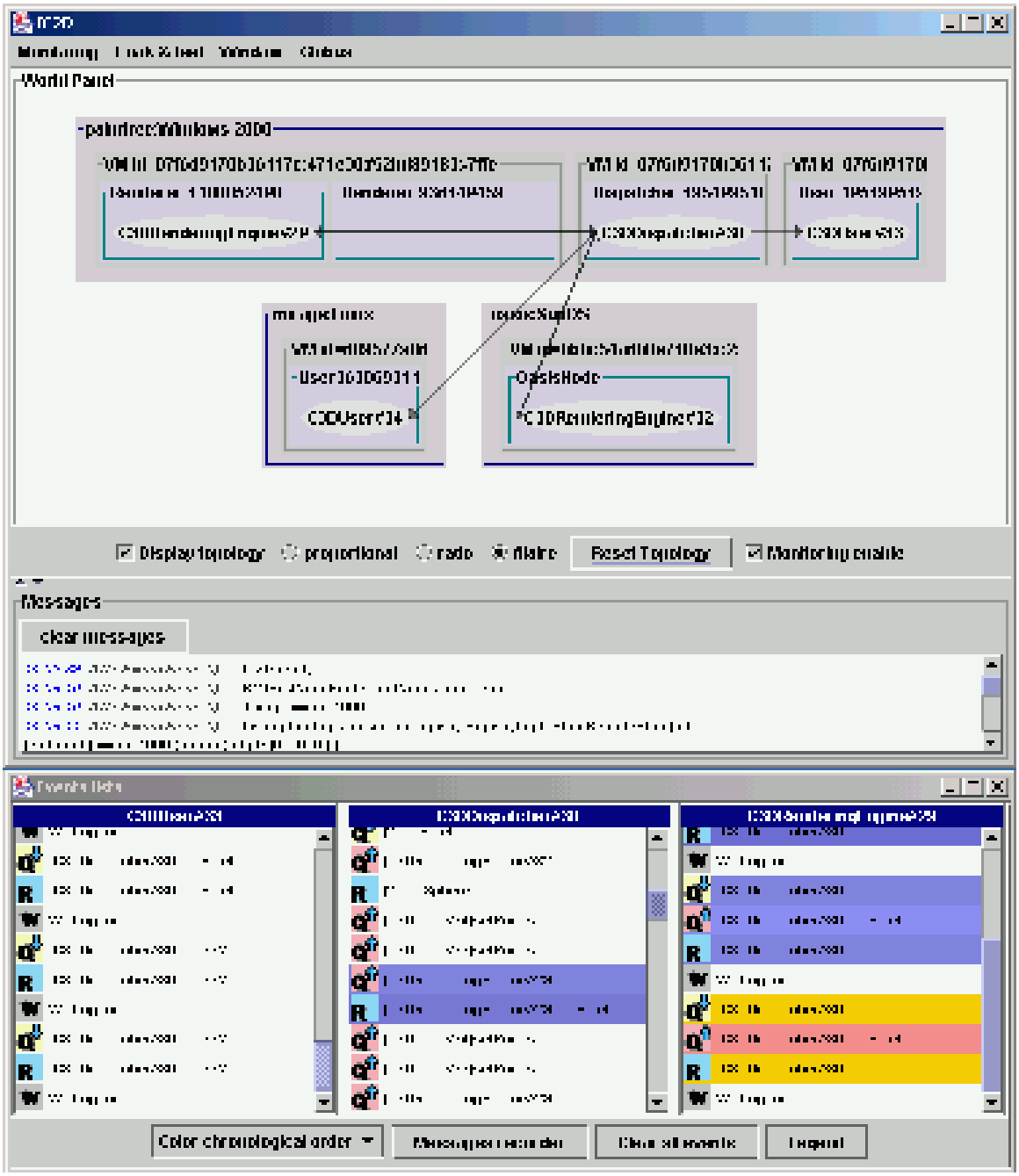
Graphical Visualisation	Textual Visualisation	Control and Monitoring
Hosts, Java Virtual Machines, Active Objects	Ordered list of messages	Interactive control of mapping upon creation
Topology: reference and communications	Status: waiting for a request for a data	Interactive control of destination upon migration
Status of active objects (executing, waiting, etc.)	Causal dependencies between messages	Dynamic change of deployment
Migration of activities	Related events (corresponding send and receive, etc.)	Drag and Drop migration of executing tasks

The full source of IC2D is included in the distribution and is also browseable on line.

Source code index [../doc/ProActive_src_html/index.xml]

25.1. Graphical Visualisation within IC2D

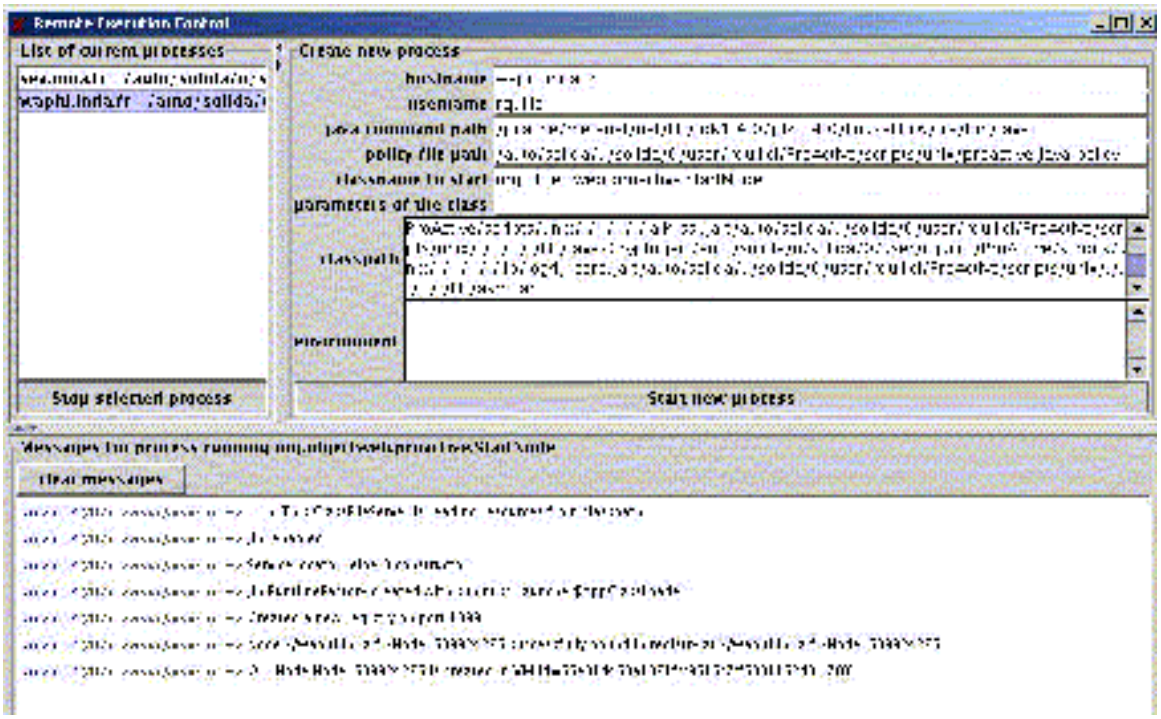
IC2D features graphical visualisation of hosts, Java Virtual Machines, and active objects, including the topology and volume of communications.



ic2d_c3d.jpg

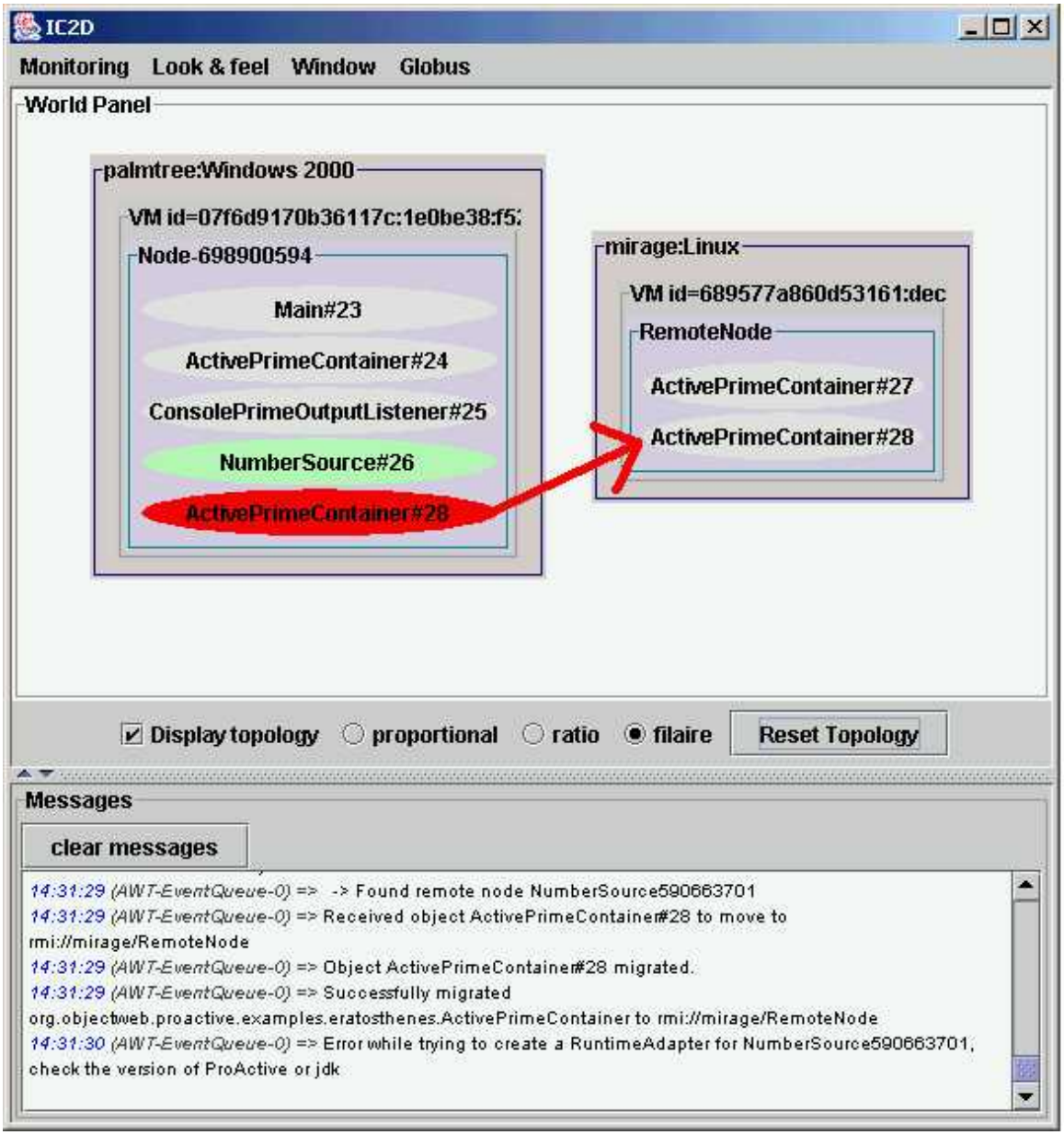
25.2. Control within IC2D

IC2D permits to interactively and dynamically create new Jvms and Nodes.



RemoteControl.jpg

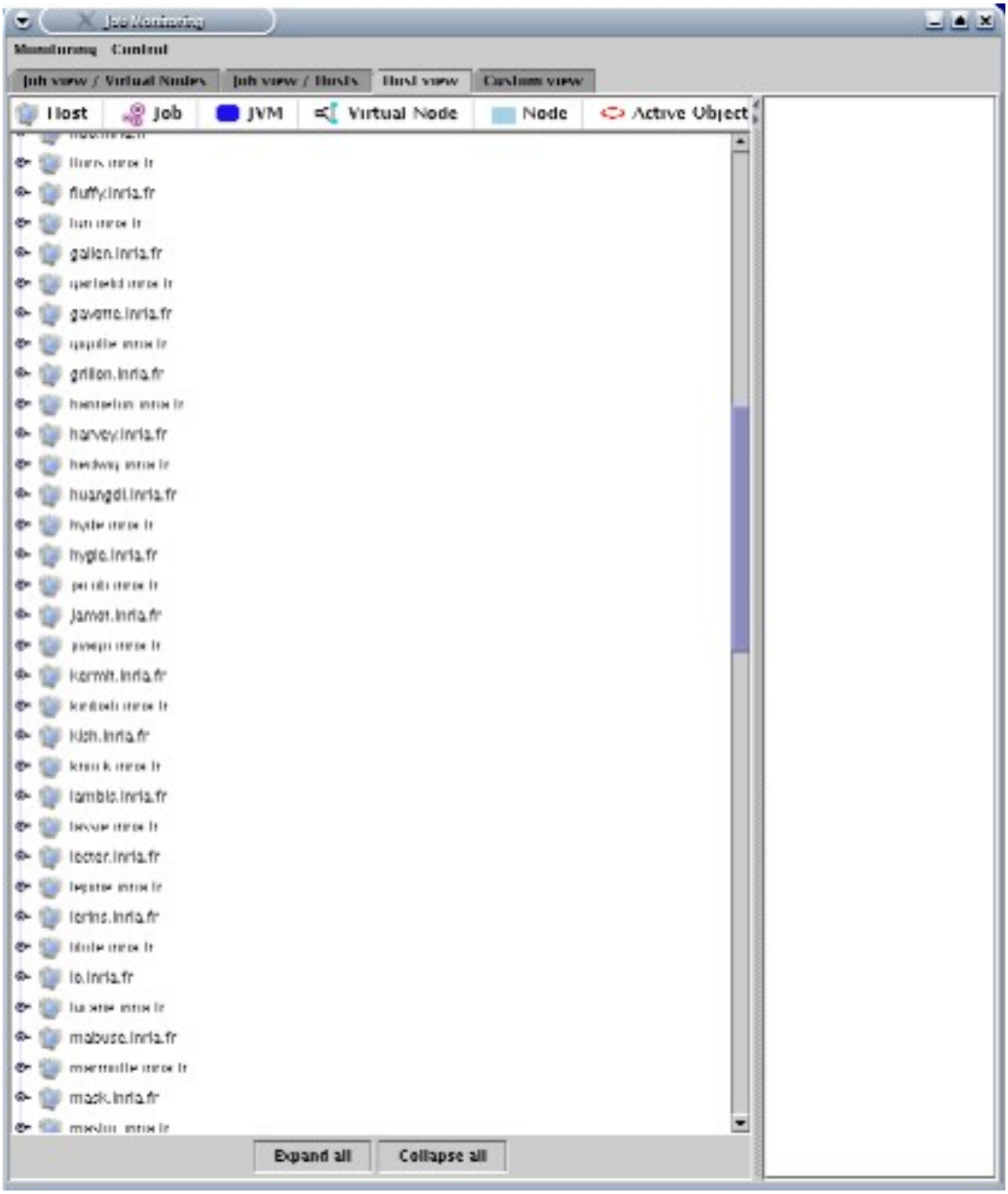
A 'drag-and-drop' migration allows to graphically move running active objects between machines.

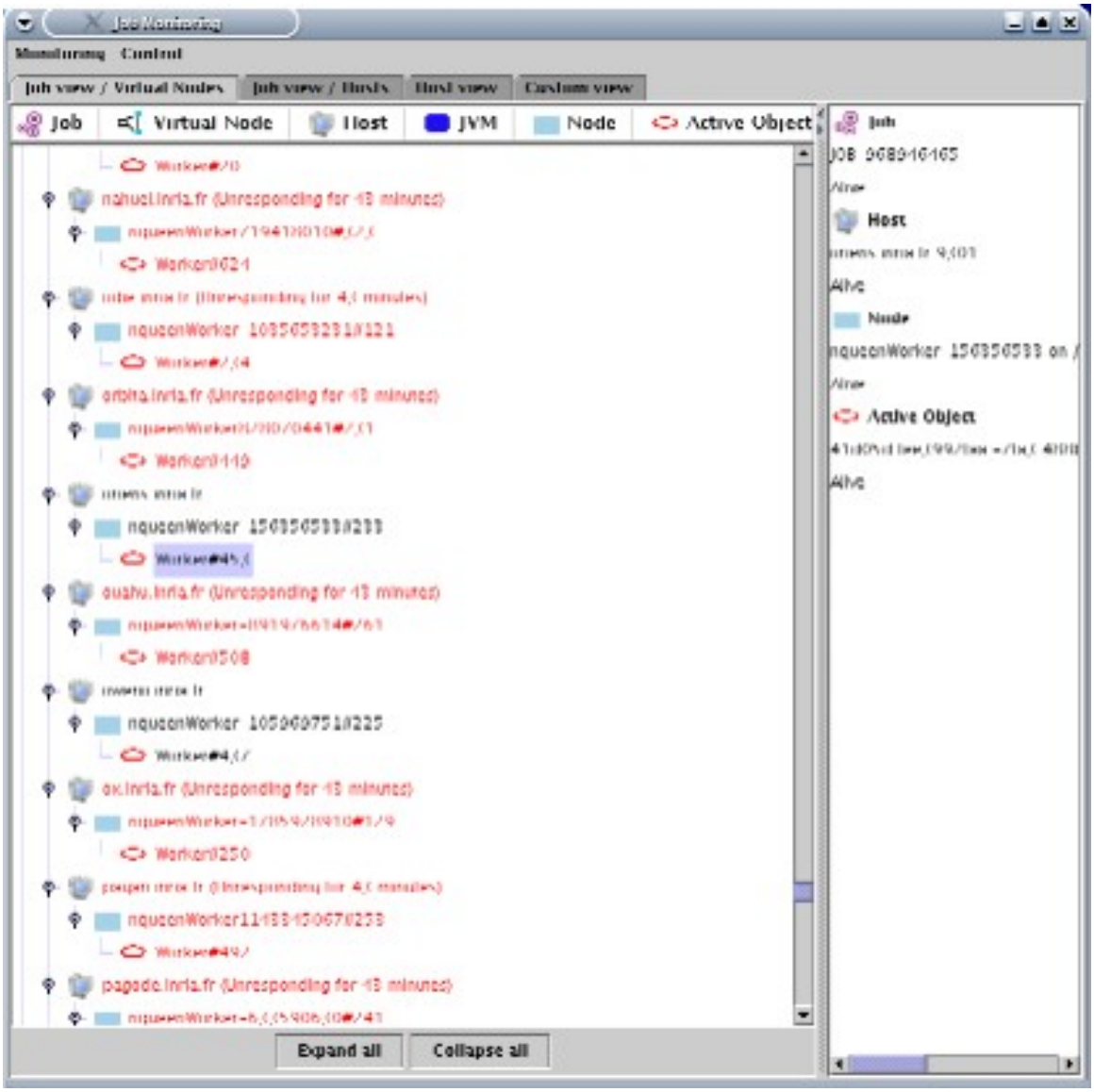


IC2D permits to visualize related events and AOs state

25.3. Job monitoring and control

IC2D now provides a Job monitoring tool, in order to visualize and control high number of resources. This tool offers different views of a deployed application: per hosts, per jobs,.... To start the Job monitor tool, choose in the IC2D menu: windows --> Show Job Monitor windows


Monitoring
Control
Job view / Virtual Nodes
Job view / Hosts
Host view
Custom view
Host
Job
JVM
Virtual Node
Node
Active Object
fluffy.inria.fr
galion.inria.fr
gavone.inria.fr
grillon.inria.fr
harvey.inria.fr
huangdi.inria.fr
hygie.inria.fr
janet.inria.fr
kirsch.inria.fr
lambic.inria.fr
locher.inria.fr
lorins.inria.fr
lo.inria.fr
mabuse.inria.fr
mack.inria.fr
Expand all
Collapse all



25.4. Launcher

25.4.1. Principles

The launcher allows users to launch applications directly from an XML descriptor file, without any script. The new XML descriptor is nearly the same as classical descriptor files, the syntax is only extended. The deployment will be done in two different phasis.

first, a new node, a "main node" will be created and activated and then, it is this node that will deploy the rest of the application.

25.4.2. MainDefinition tag

A new tag has been introduced, just before the component definition tag. This tag is named "mainDefinition" and its syntax is :

```
<mainDefinition id="mainID" class="theClassToLaunchContainingAMainMethod">
```

```
  <arg value="param1">
  <arg value="param2">
  <mapToVirtualNode value="main-Node"/>
 </mainDefinition>
```

Eventually, several mains might be defined so the **id** allows to identify all mainDefinitions.

The **class** attribute is the path where can be found the class to launch.

This class MUST contain a main method.

Then any number of parameters can be declared in **arg** tags. The parameters will be given to the main method in the same order the were declared.

And finally a **mapToVirtualNode** tag will link the main info to virtual node, declared with the same name in the virtualNodeDefinitions tag (in componentDefinition).

```
 <?xml version="1.0" encoding="UTF-8"?>
 <ProActiveDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
  <!-- <security file="../../descriptors/c3dPolicy.xml"></security> -->
  < componentDefinition>
   < virtualNodesDefinition>
    < virtualNode name="Dispatcher"
property="unique_singleAO"/>
    < virtualNode name="Renderer"/>
   < /virtualNodesDefinition>
  < /componentDefinition>
  < deployment>
   < register virtualNode="Dispatcher"/>
   < mapping>
    < map virtualNode="Dispatcher">
     < jvmSet>
      < currentJVM/>
     < /jvmSet>
    < /map>
    < map virtualNode="Renderer">
     < jvmSet>
     < vmName value="Jvm1"/>
     < vmName value="Jvm2"/>
     < vmName value="Jvm3"/>
     < vmName value="Jvm4"/>
     < /jvmSet>
    < /map>
   < /mapping>
   < jvms>
    < jvm name="Jvm1">
     < creation>
      < processReference
refid="localJVM"/>
     < /creation>
    < /jvm>
    < jvm name="Jvm2">
     < creation>
```

```
      < processReference
refid="localJVM"/>
     < /creation>
    < /jvm>
     < jvm name="Jvm3">
     < creation>
       < processReference refid="localJVM"/>
     < /creation>
    < /jvm>
    < jvm name="Jvm4">
     < creation>
      < processReference refid="localJVM"/>
     < /creation>
    < /jvm>
   < /jvms>
  < /deployment>
  < infrastructure>
   < processes>
    < processDefinition id="localJVM">
     < jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess"></jvmProcess>
    < /processDefinition>
   < /processes>
  < /infrastructure>
 </ProActiveDescriptor>
```

25.4.3. API

The Launcher class is located in the package **org.objectweb.proactive.core.descriptor**. To use it you will have to create a new instance of the launcher with the path of the XML descriptor (this descriptor must contain a **mainDefinition tag**). The constructor will parse the file and reify a ProActiveDescriptor. You only have to call the **activate()** method on the launcher instance to launch the application.

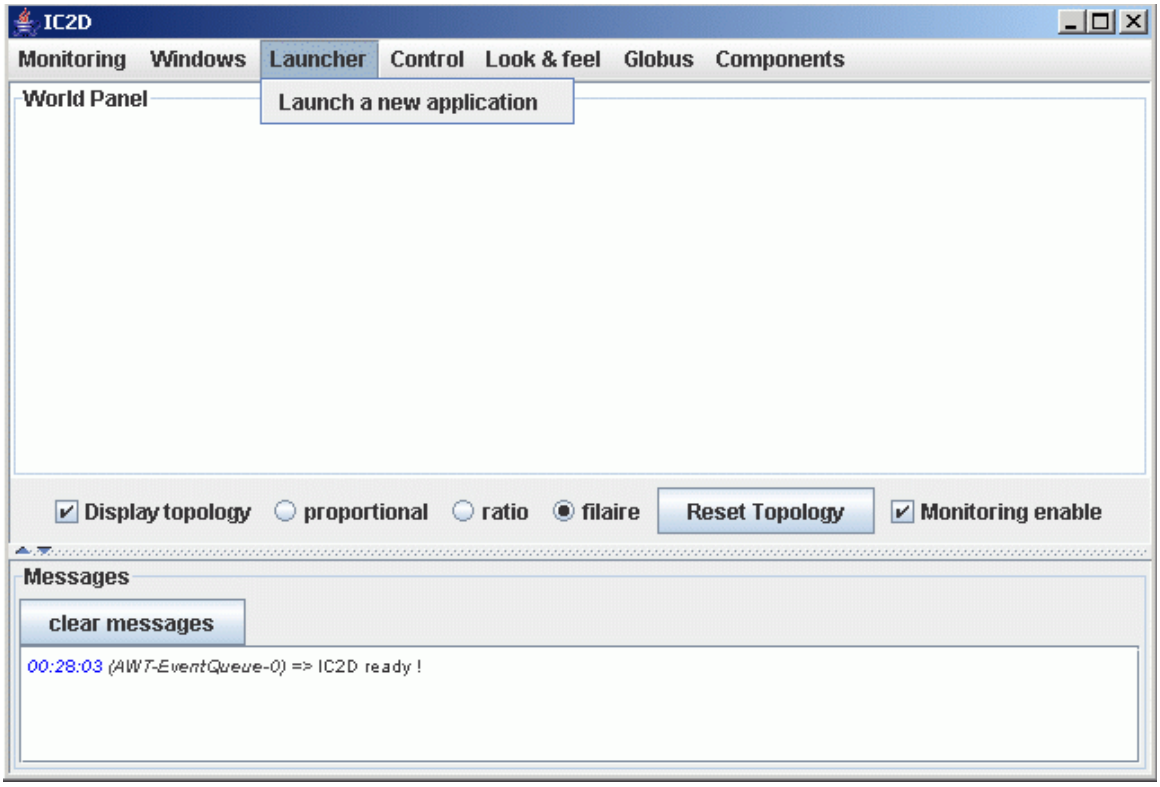
For example:
```
Launcher launcher = new Launcher ("myDescriptor.xml") ;

launcher.activate() ;
```

you can also get the ProActiveDescriptor built by the launcher by calling the getDescriptor() method on the launcher instance.

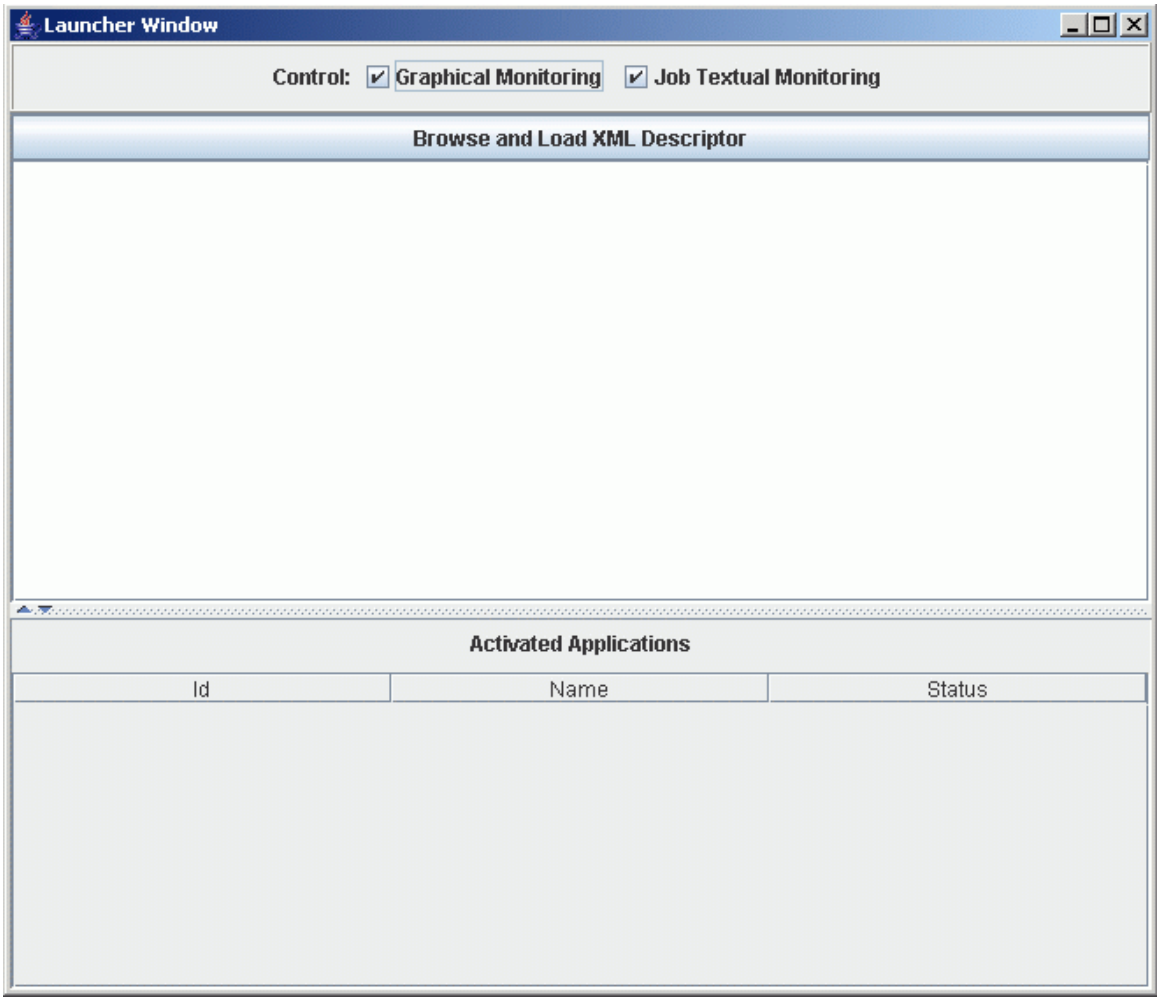
```
ProActiveDescriptor pad = launcher.getDescriptor() ;
```

25.4.4. Launcher in IC2D

You will find the launcher in the **launcher menu**, in the menu bar. Click then on **launch a new application**.



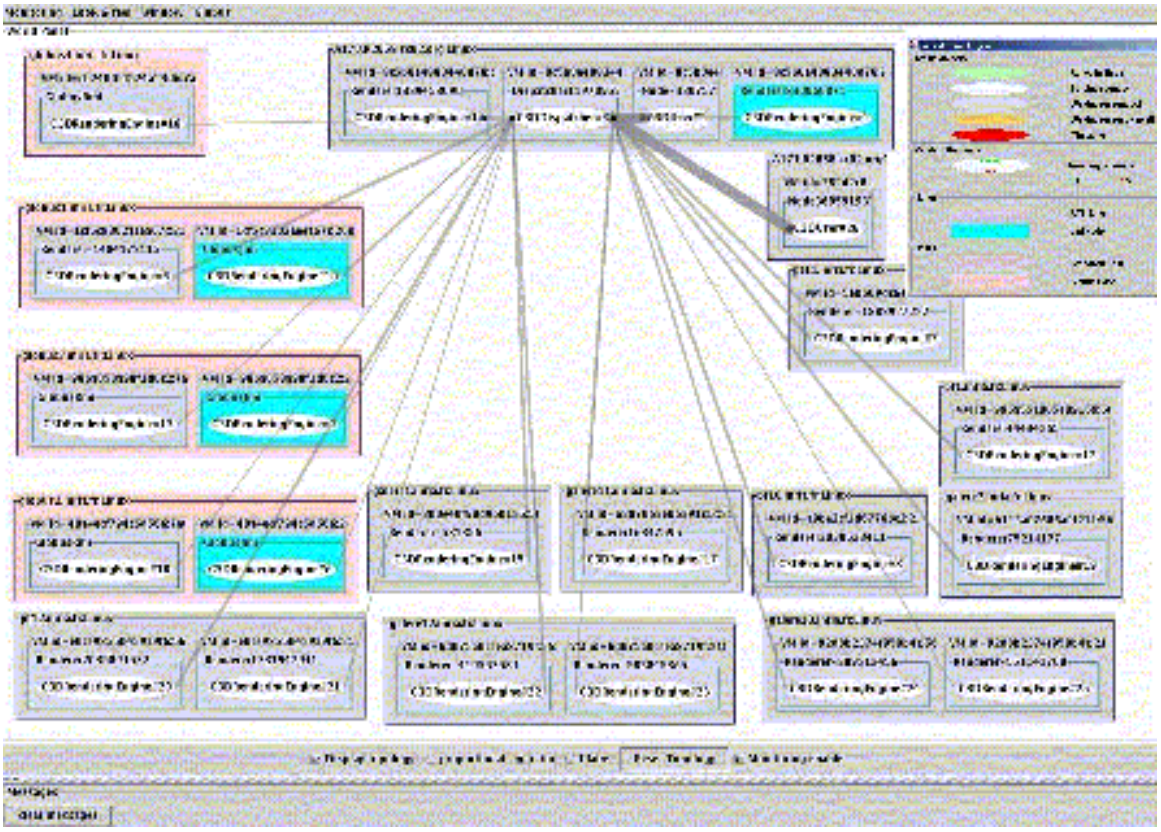
Now it opens a new window proposing you to browse your file system for a XML descriptor and load it into a descriptors list. A right click on the list items allows you to launch (activate) the application. Two checkbox control the graphical monitoring and textual job monitoring your application.



You can kill the applications launched with from a popupmenu in the "activated applications" table.

25.5. Grid and cluster computing

As ProActive is interfaced with Globus and LSF, IC2D permits to interactively control and debug applications that execute on intercontinental-wide networks. Below is C3D application deployed with JINI and RMI protocol between Baltimore US and Nice France on a globus cluster and LSF cluster



This experiment was partly sponsored by

Sun microsystems

Chapter 26. ProActive Eclipse plugin

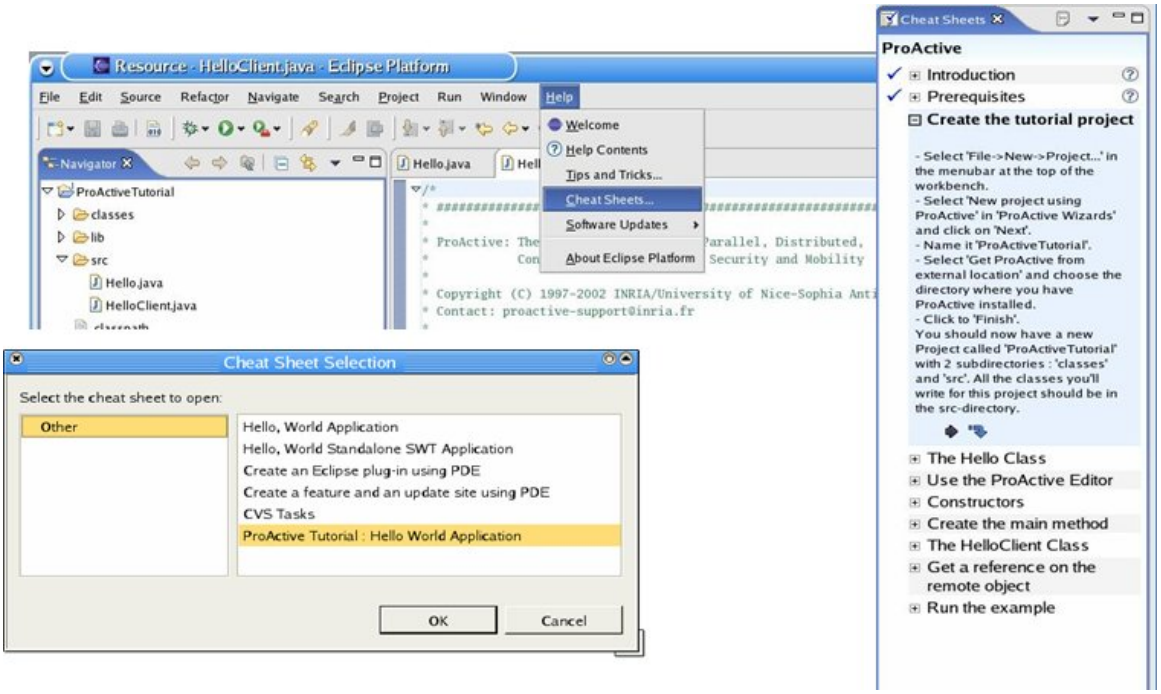
26.1. Overview

The main objective of this plugin is to facilitate the use of ProActive in the Eclipse environment. The plugin has three main purposes :

- ***A ProActive guided Tour***
- ***ProActive Wizards***
- ***A ProActive editor***

26.2. The Guided Tour

The aim of the guided tour is to provide a step by step explanation to the ProActive beginners.



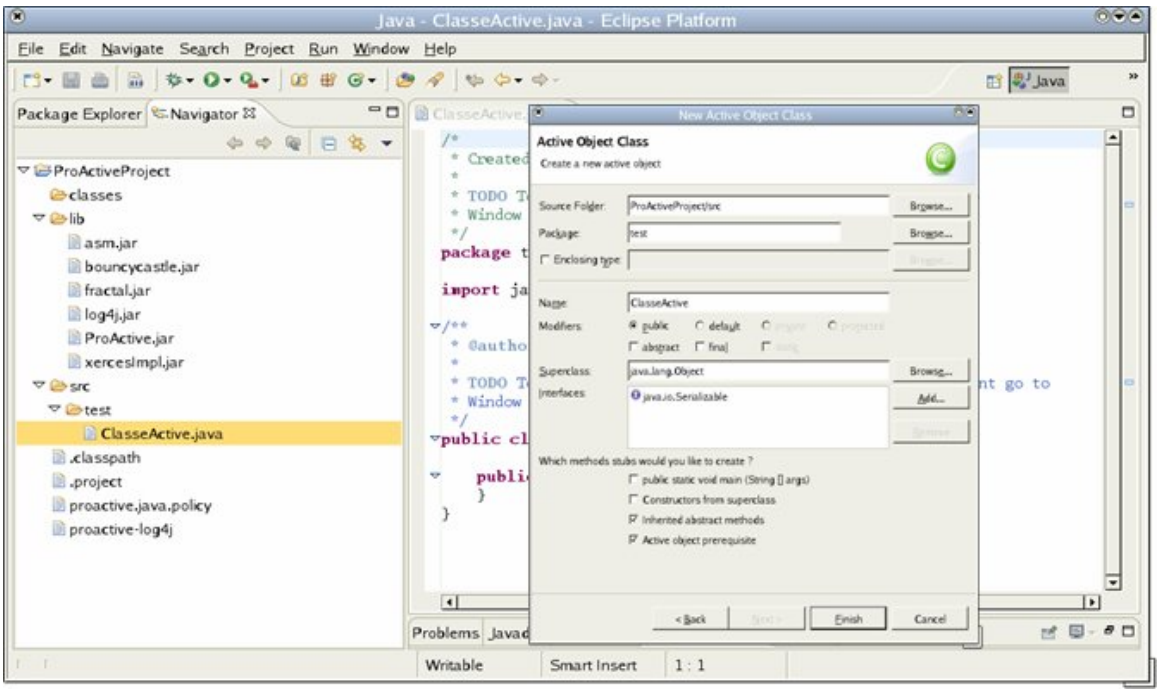
This guided tour (that is actually eclipse cheat sheet) purposes :

- To Explain ProActive to beginners
- To make interactions with the user with simple situations
- To Show the important points

26.3. Wizards

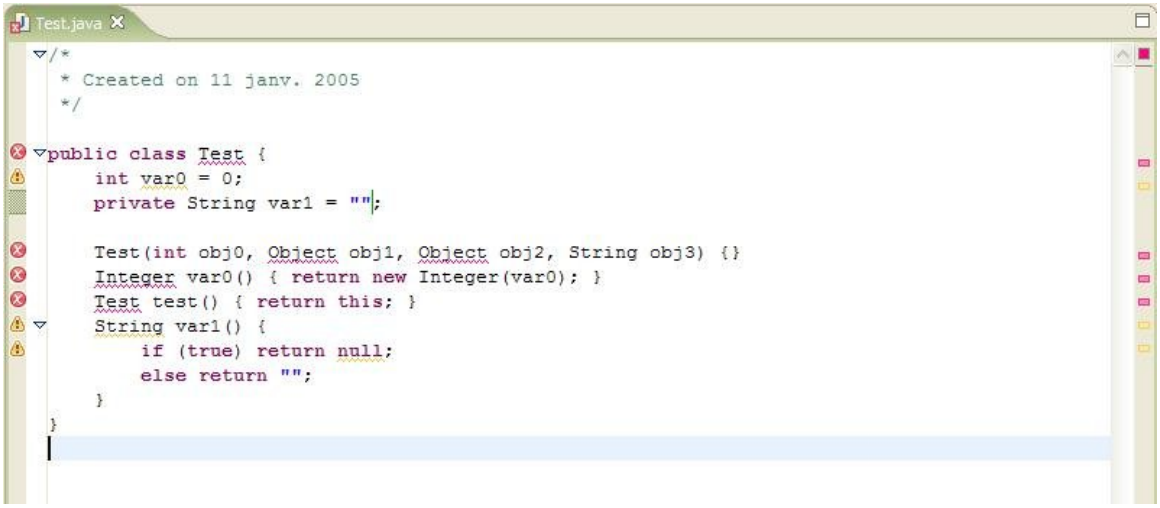
These wizards will guide developpers to make complex operations with ProActive, such as installation, integration , configuration or execution. We have created 4 wizards :

1. a ProActive installation wizard
2. a wizard that create applications using ProActive
3. an active object creation wizard
4. a configuration and execution wizard



26.4. The ProActive Editor

This editor checks coding rules. It informs the developper of error concerning ProActive in his classes and can resolve some of these errors.



Part VII. Guided Tour

Table of Contents

Chapter 27. ProActive guided tour

This tour is a practical introduction to ProActive, giving an illustrated introduction to some of the functionnalities and facilities offered by the library.

First off, we give an explaination on how to install and configure ProActive. Next are introduced the different functionalities of the library through some runnning examples. Then are given om details on how this is put down in code, and you will be challenged to write your bits of code. This should give you practical experience on how to program using ProActive. A section on OO-SPMD (Object-Oriented Single Program Multiple Data) will show how to use this programming paradigm.

The guided tour is finished off by the complete N-Body example. This application is first written trivially, then some speed-ups are plugged in, to show how ProActive can help you. We hope this will help your understanding of the library, and of the concepts driving it.

If you need further details on how the examples work, check the ProActive applications [http://www-sop.inria.fr/oasis/ProActive/apps/index.xml] page.

27.1. Installation and setup

Follow the instructions [http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/Installation.xml] for downloading and installing ProActive.

The programming exercises in the first part imply that you :

- Don't forget to add the required libraries to your classpath (i.e. the libraries contained in the ProActive/lib directory, as well as either the proactive.jar archive, or the compiled classes of proactive (better if you modify the source code)
- use a policy file, such as ProActive/scripts/proactive.security.policy, with the JVM option -Djava.security.policy=/filelocation/proactive.java.policy

Set the CLASSPATH as follow :

Under linux :

```
export CLASSPATH=.:./ProActive_examples.jar:./ProActive.jar:./lib/bcel.jar:./lib/asm.jar:./lib/log4j.jar:./lib/xercesImpl.jar:./lib/cor
```

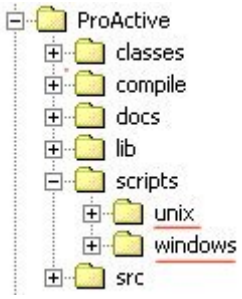
Under windows :

```
set CLASSPATH=.;.\ProActive_examples.jar;.\ProActive.jar;.\lib\bcel.jar;.\lib\asm.jar;.\lib\log4j.jar;.\lib\xercesImpl.jar;.\lib\compo
```

Concerning the second part of the tutorial (examples of some functionalities):

- Note that the compilation is managed by Ant [http://jakarta.apache.org/ant] ; we suggest you use this tool to make modifications to the source code, while doing this tutorial. Nevertheless, you can just change the code and compile using compile.sh <the example application> (or compile.bat under windows)
- The examples used in the second part of this tutorial are provided in the /scripts directory of the distribution.

  The scripts are platform dependant : .sh files on linux are equivalent to the .bat files on windows


ProActive
classes
compile
docs
lib
scripts
unix
windows
src

Chapter 28. Introduction to some of the functionalities of ProActive

This chapter will present some of the facilities offered by ProActive, namely :

- parallel processing : how you can run several tasks in parallel.
- synchronization : how you can synchronize tasks.
- migration : how you can migrate Active Objects.

28.1. Parallel processing and collaborative application with ProActive

Distribution is often used for CPU-intensive applications, where parallelism is a key for performance.

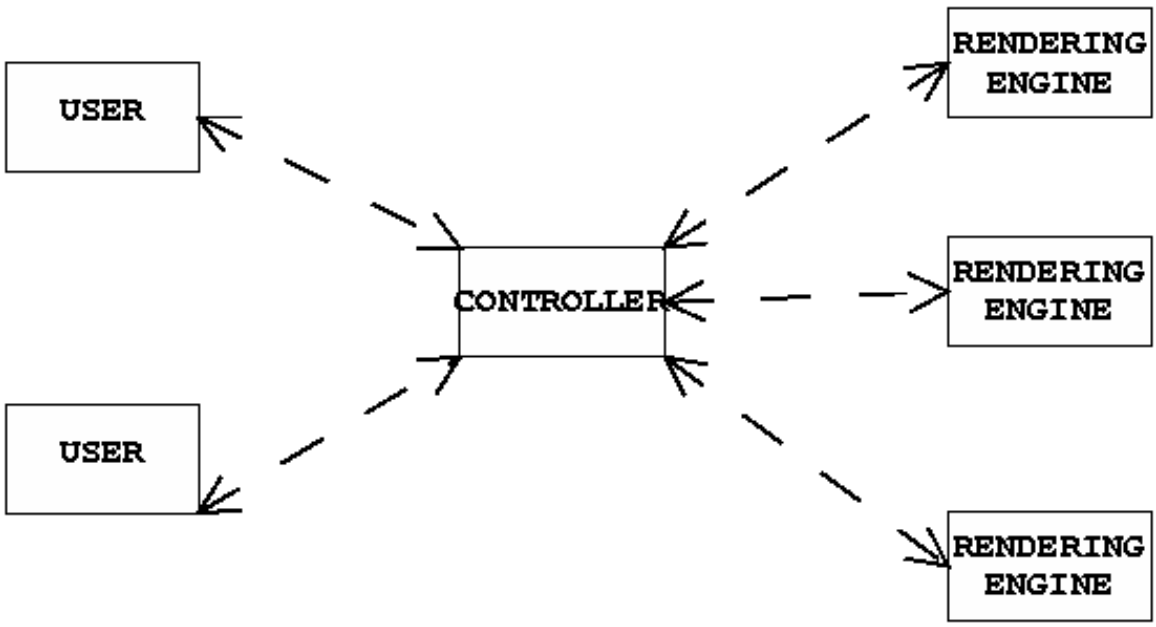
A typical application is C3D.

Note that parallelisation of programs can be facilitated with ProActive, thanks to asynchronism method calls [../FutureObjectCreation.xml], as well as group communications [../TypedGroupCommunication.xml].

28.2. C3D : a parallel, distributed and collaborative 3D renderer

C3D [http://www-sop.inria.fr/oasis/ProActive/apps/c3d.xml] is a Java benchmark application that measures the performance of a 3D raytracer renderer distributed over several Java virtual machines using Java RMI. It showcases some of the benefits of ProActive, notably the ease of distributed programming, and the speedup through parallel calculation.

Several users can collaboratively view and manipulate a 3D scene. The image of the scene is calculated by a dynamic set of rendering engines using a raytracing algorithm, everything being controlled by a central dispatcher.

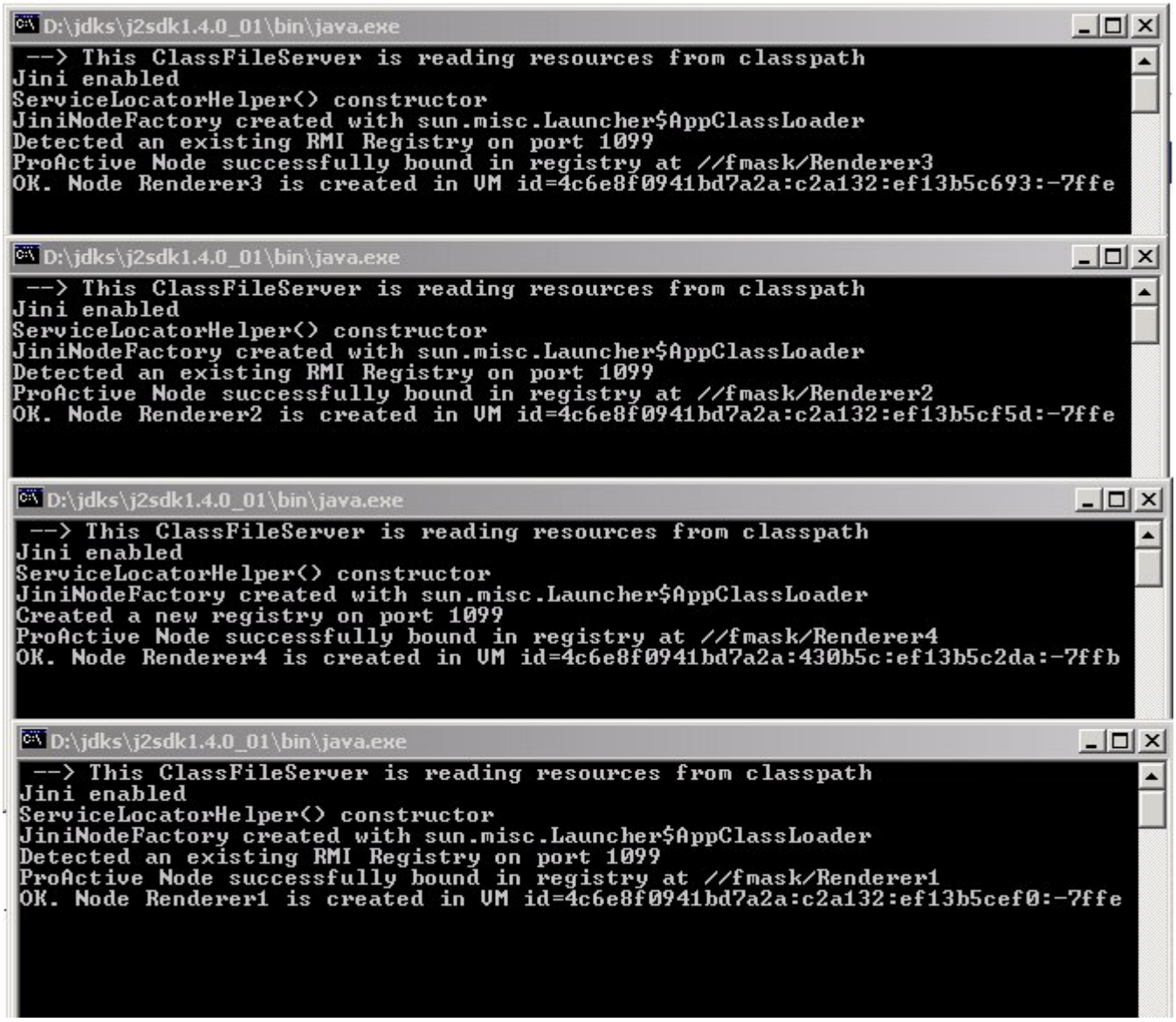


the active objects in the c3d application

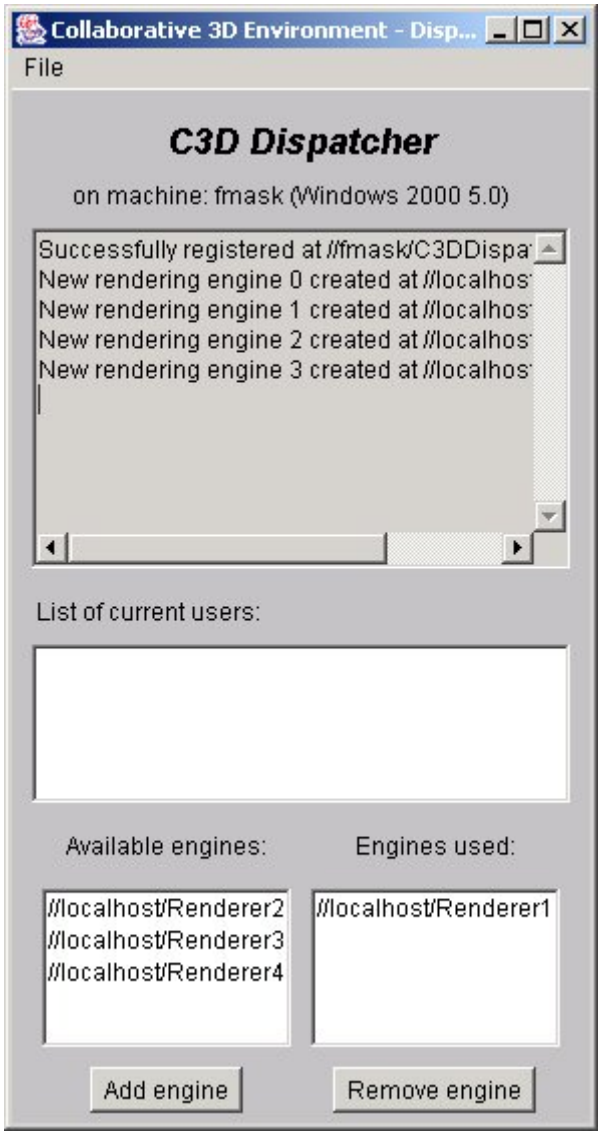
28.2.1. 1. start C3D

using the script c3d_no_user

A "Dispatcher" object is launched (ie a centralized server) as well as 4 "Renderer" objects, that are active objects to be used for parallel rendering.



the 4 renderers are launched



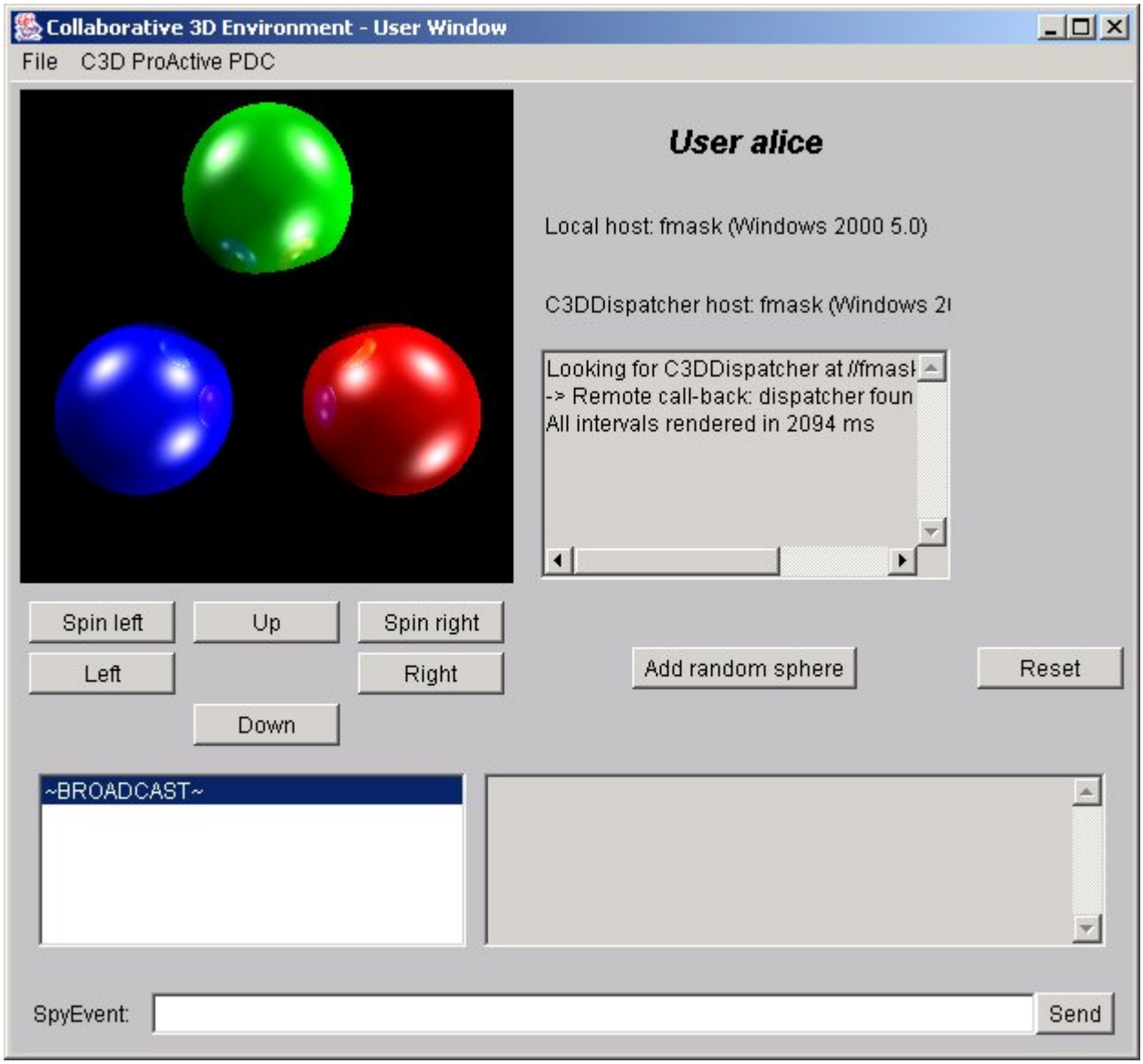
the dispatcher GUI is launched

The bottom part of the window allows the addition and removal of renderers.

28.2.2. 2. start a user

using c3d_add_user

- connect on the current host (proposed by default) by just giving your name



for example the user "alice"

- spin the scene, add a random sphere, and observe how the action takes place immediately

- add and remove renderers, and observe the effect on the "speed up" indication from the user window.

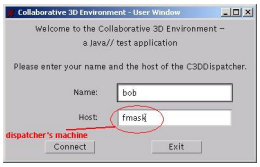
Which configuration is the fastest for the rendering?

Are you on a multi-processor machine?

** you might not perceive the difference of the performance. The difference is better seen with more distributed nodes and objects (for example on a cluster with 30+ renderers).*

28.2.3. 3. start a user from another machine

using the c3d_add_user script, and specifying the host (NOT set by default)



If you use rlogin, make sure the DISPLAY is properly set.

You must use the same version of ProActive on both machines!

- test the collaborative behavior of the application when several users are connected.

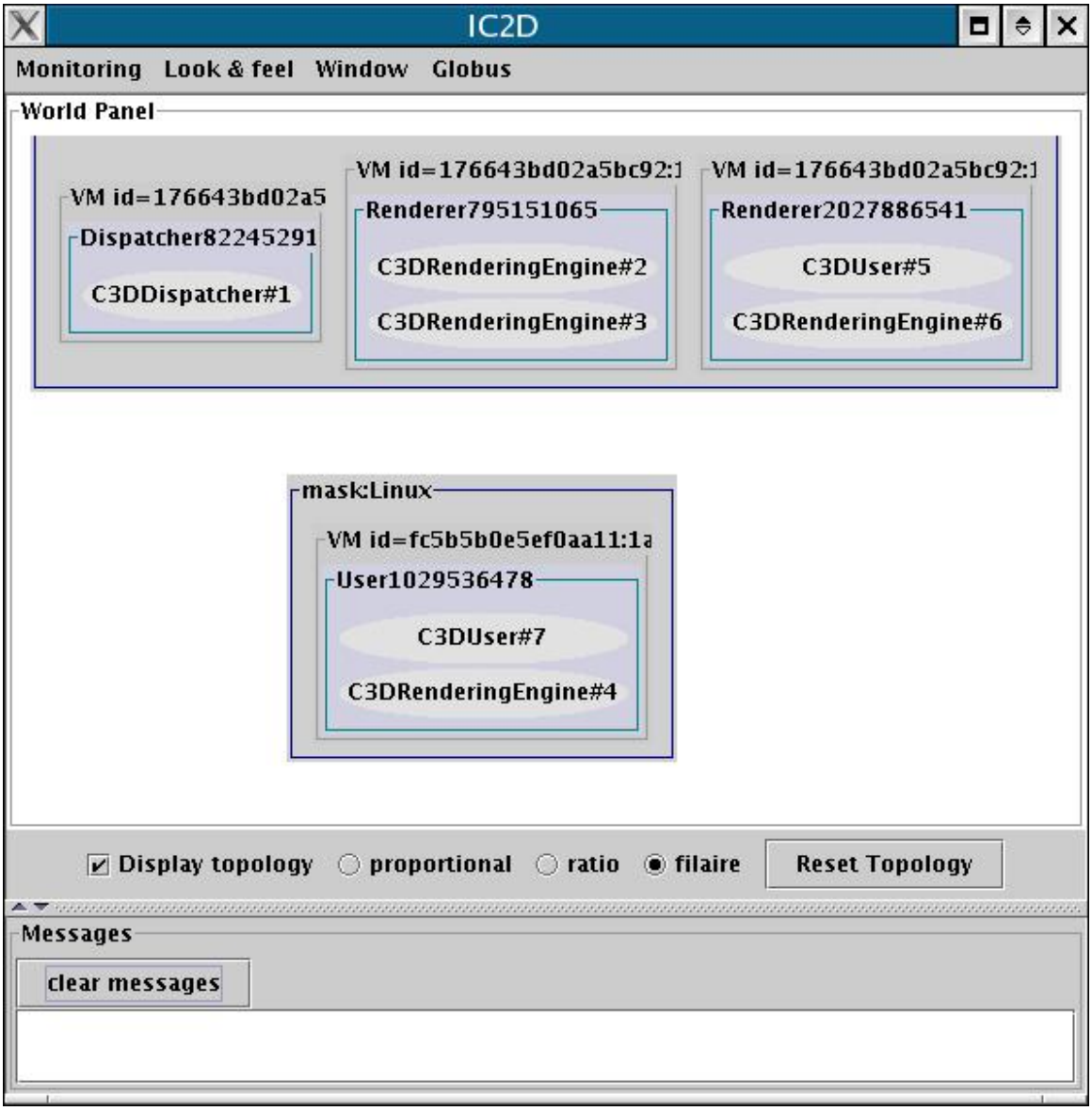
Notice that a collaborative consensus must be reached before starting some actions (or that a timeout occured).

28.2.4. 4. start IC2D to visualize the topology

- to visualize all Active objects, you need to acquire ("monitoring" menu) :

- the machine on which you started the "Dispatcher"

- the machine on which you started the second user



- add random spheres for instance, and observe messages (Requests) between Active Objects.

- add and remove renderers, and check graphically whether the corresponding Active Objects are contacted or not, in order to achieve the rendering.

- you can textually visualize this information by activating "add event timeline for this WorldObject" on the World panel with the right mouse button, and then "show the event list window" on the top menu window

28.2.5. 5. drag-and-drop migration

- from IC2D, you can drag-and-drop active objects from one JVM to another. Click the right button on a C3DRenderingEngine, and drag and drop it in another JVM. Observe the migration taking place.

- add a new sphere, using all rendering engines, and check that the messages are still sent to the active object that was asked to migrate.

- as migration and communications are implemented in a fully compatible manner, you can even migrate with IC2D an active object while it is communicating (for instance when a rendering action is in progress). Give it a try!

Since version 1.0.1 of the C3D example, you can also migrate the client windows!

28.2.6. 6. start a new JVM in a computation

manually you can start a new JVM - a "Node" in the ProActive terminology - that will be used in a running system.

- on a different machine, or by remote login on another host, start another Node, named for instance NodeZ :

> under linux :`startNode.sh rmi://mymachine/NodeZ & (or startNode.bat rmi://mymachine/NodeZ)`

The node should appear in IC2D when you request the monitoring of the new machine involved (Monitoring menu, then "monitor new RMI host".

- the node just started has no active object running in it. Drag and drop one of the renderers, and check that the node is now taking place in the computation :

> - spin the scene to trigger a new rendering
>
> - see the topology

** if you feel uncomfortable with the automatic layout, switch to manual using the "manual layout" option (right click on the World panel). You can then reorganize the layout of the machines.*

- to fully distribute the computation, start several nodes (you need 2 more) and drag-and-drop renderers in them.

Depending on the machines you have, the complexity of the image, look for the most efficient configuration.

28.2.7. 7. have a look at the source code for the main classes of this application :

`org.objectweb.proactive.examples.c3d.C3DUser.java`

`org.objectweb.proactive.examples.c3d.C3DRenderingEngine.java`

`org.objectweb.proactive.examples.c3d.C3DDispatcher.java`

> look at the method public void processRotate(org.objectweb.proactive.Body body, String methodName, Request r) that handles election of the next action to undertake.

28.3. Synchronization with ProActive

ProActive provides an advanced synchronization mechanism that allows an easy and safe implementation of potentially complex synchronization policies.

This is illustrated by two examples :

- The readers and the writers
- The dining philosophers

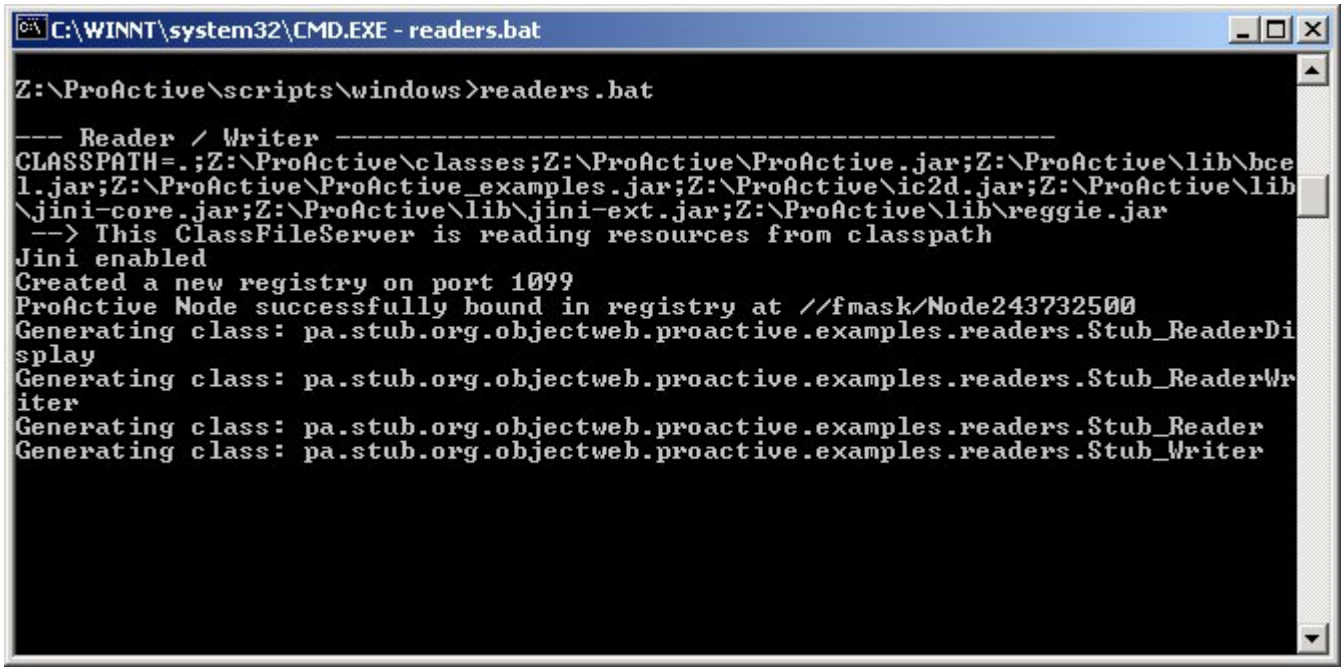
28.3.1. The readers-writers

The readers and the writers want to access the same data. In order to allow concurrency while ensuring the consistency of the readings, accesses to the data have to be synchronized upon a specified policy. Thanks to ProActive, the accesses are guaranteed to be allowed sequentially.

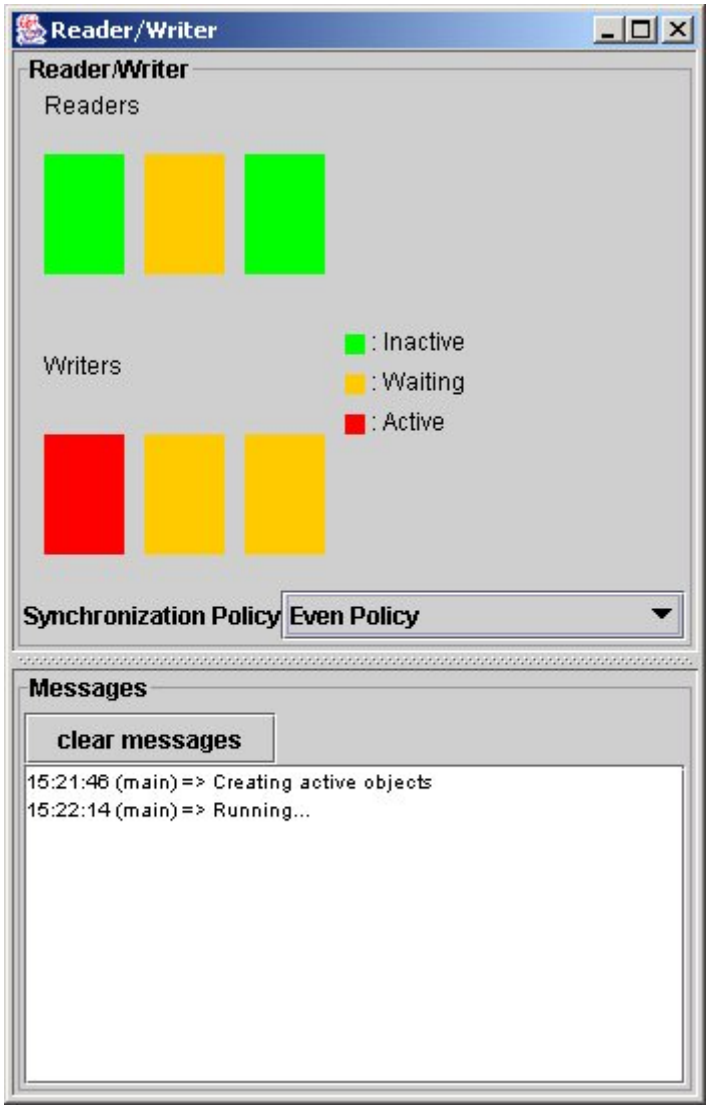
The implementation with ProActive [http://www-sop.inria.fr/oasis/ProActive/apps/readers.xml] uses 3 active objects : Reader, Writer, and the controller class (ReaderWriter).

28.3.1.1. 1. start the application

using the readers script



ProActive starts a node (i.e. a JVM) on the current machine, and creates 3 Writer, 3 Reader, a ReaderWriter (the controller of the application) and a ReaderDisplay, that are active objects.



a GUI is started that illustrates the activities of the Reader and Writer objects.

28.3.1.2. 2. look and check the effect of different policies : even, writer priority, reader priority

What happens when priority is set to "reader priority" ?

28.3.1.3. 3. look at the code for programming such policies

in `org.objectweb.proactive.examples.readers.ReaderWriter.java`

More specifically, look at the routines in :

public void evenPolicy(org.objectweb.proactive.Service service)

public void readerPolicy(org.objectweb.proactive.Service service)

public void writerPolicy(org.objectweb.proactive.Service service)

Look at the inner class `MyRequestFilterm` that implements `org.objectweb.proactive.core.body.request.Requ`

How does it work?

28.3.1.4. 4. Introduce a bug in the Writer Priority policy

For instance, let several writers go through at the same time.

- observe the Writer Policy policy before recompiling

- recompile (using compile.sh readers or compile.bat readers)

- observe that stub classes are regenerated and recompiled

- observe the difference due to the new synchronization policy : what happens now?

- correct the bug and recompile again ; check that everything is back to normal

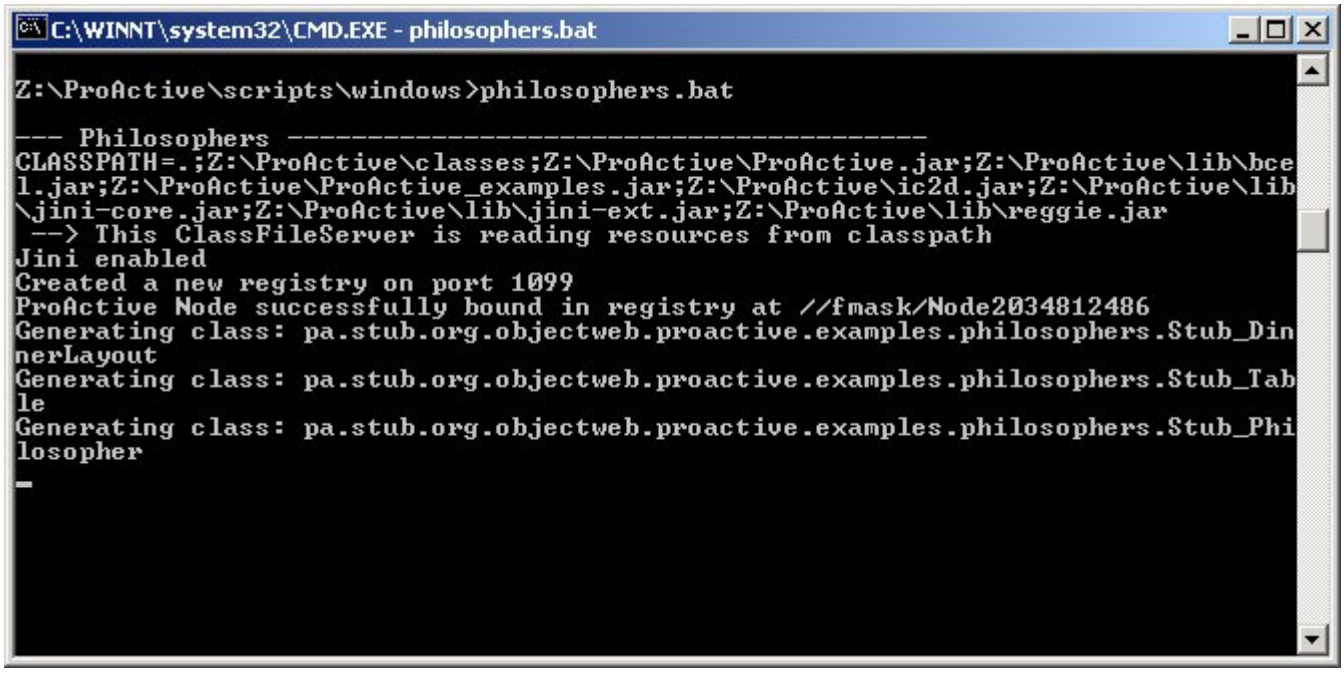
28.3.2. The dining philosophers

The "dining philosophers" problem is a classical exercise in the teaching of concurrent programming. The goal is to avoid deadlocks.

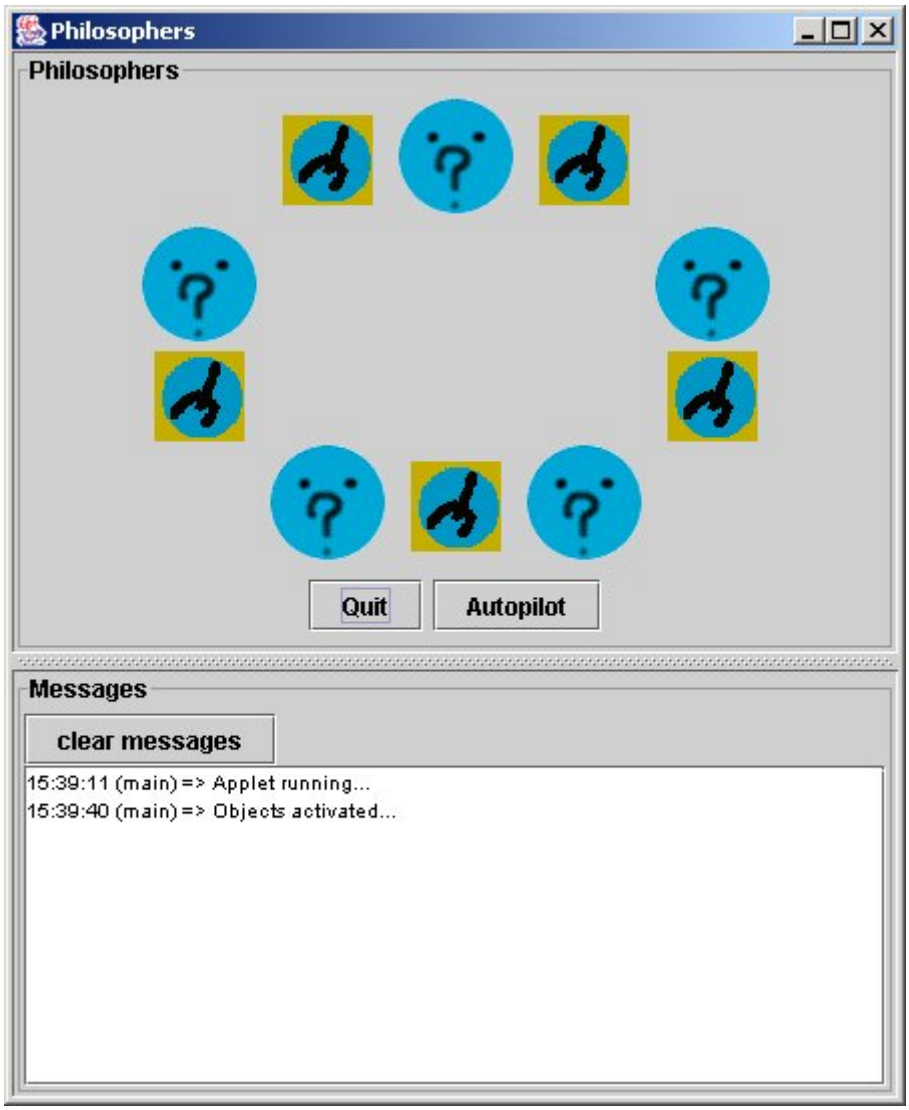
We have provided an illustration of the solution [http://www-sop.inria.fr/oasis/ProActive/apps/phil.xml] using ProActive, where all the philosophers are active objects, as well as the table (controller) and the dinner frame (user interface).

28.3.2.1. 1. start the philosophers application

with philosophers.sh or philosophers.bat



ProActive creates a new node and instantiates the active objects of the application : DinnerLayout, Table, and Philosopher



the GUI is started.

28.3.2.2. 2. understand the color codes

Philosophers **philosophing**

hungry, wants the fork !

eating

Forks **taken**

free

28.3.2.3. 3. test the autopilot mode

The application runs by itself without encountering a deadlock.

28.3.2.4. 4. test the manual mode

Click on the philosophers' heads to switch their modes

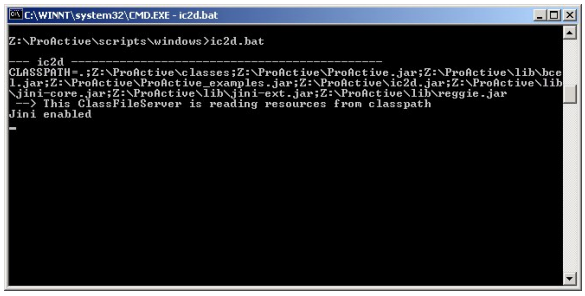
Test that there are no deadlocks!

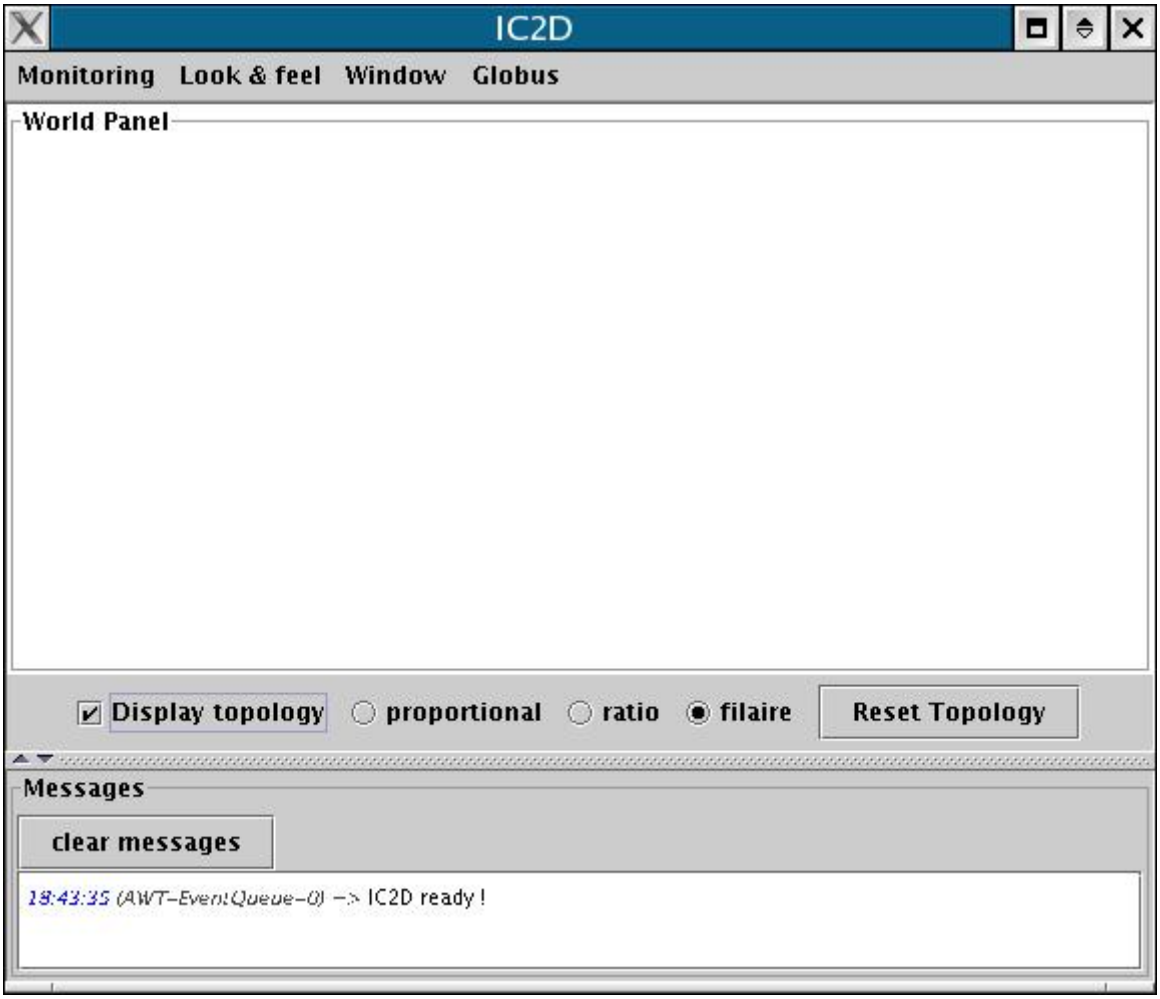
Test that you can starve one of the philosophers (i.e. the others alternate eating and thinking while one never eats!)

28.3.2.5. 5. start the IC2D application

IC2D [http://www-sop.inria.fr/oasis/ProActive/IC2D/index.xml] is a graphical environment for monitoring and steering of distributed and metacomputing applications.

- being in the autopilot mode, start the IC2D visualization application (using ic2d.sh or ic2d.bat)

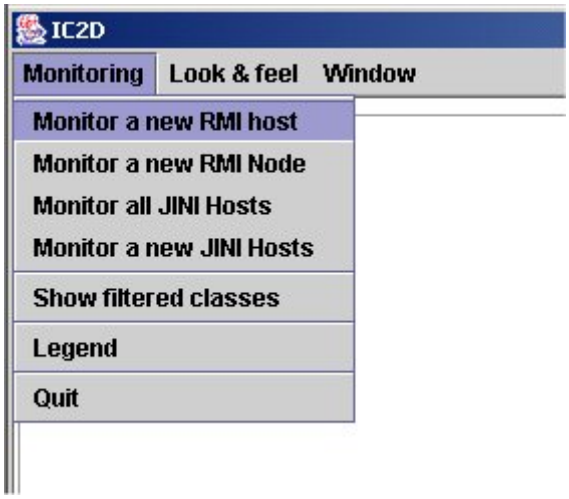




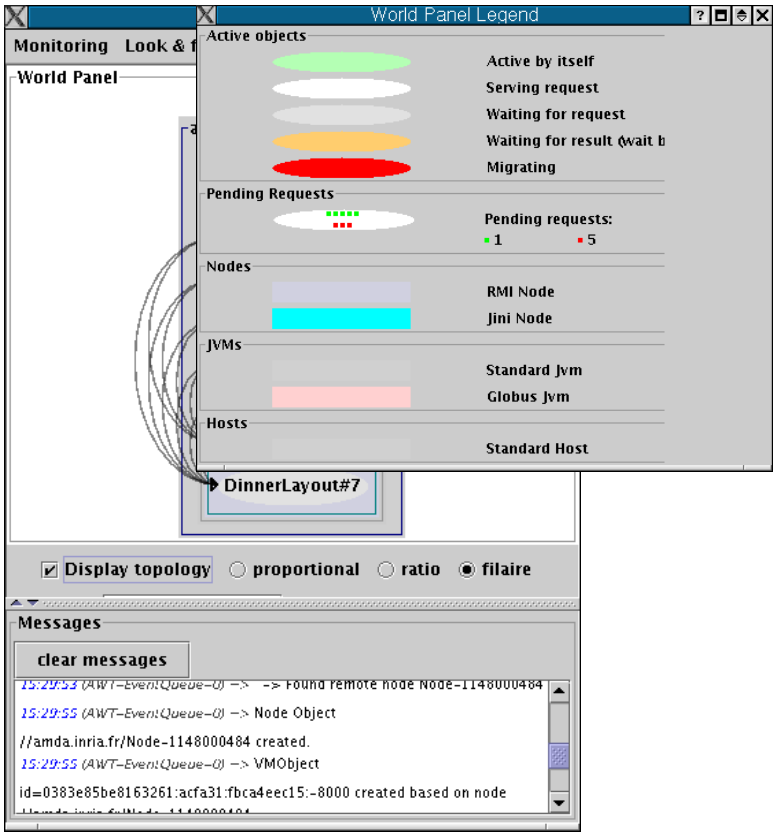
the ic2d GUI is started. It is composed of 2 panels : the main panel and the events list panel

- acquire you current machine

menu monitoring - monitor new RMI host



It is possible to visualize the status of each active object (processing, waiting etc...), the communications between active objects, and the topology of the system (here all active objects are in the same node) :



28.4. Migration of active objects

ProActive allows the transparent migration of objects between virtual machines.

A nice visual example is the penguin's one.

This example shows a set of mobile agents [http://www-sop.inria.fr/oasis/ProActive/apps/penguin.xml] moving around while still communicating with their base and with each other. It also features the capability to move a swing window between screens while moving an agent from one JVM to the other.

28.4.1. 1. start the penguin application

using the `penguin` script.

28.4.2. 2. start IC2D to see what is going on

using the `ic2d` script

acquire the machines you have started nodes on

28.4.3. 3. add an agent

- on the Advanced Penguin Controller window : button "add agent"

an agent is materialized by a picture in a java window.

- select it, and press button "start"

- observe that the active object is moving between the machines, and that the penguin window disappears and reappears on the screen associated with the new JVM.

28.4.4. 4. add several agents

after selecting them, use the buttons to :

- communicate with them ("chained calls")

- start, stop, resume them

- trigger a communication between them ("call another agent")

28.4.5. 5. move the control window to another user

- start a node on a different computer, using another screen and keyboard

- monitor the corresponding JVM with IC2D

- drag-and-drop the active object "AdvancedPenguinController" with IC2D into the newly created JVM : the control window will appear on the other computer and its user can now control the penguins application.

- still with IC2D, doing a drag-and-drop back to the original JVM, you will be able to get back the window, and control yourself the application.

Chapter 29. Hands-on programming

Here is an introduction to programming with ProActive. It should give you a first flavor, and you will be able to see some more advanced examples in the section "introduction to some functionalities of the library".

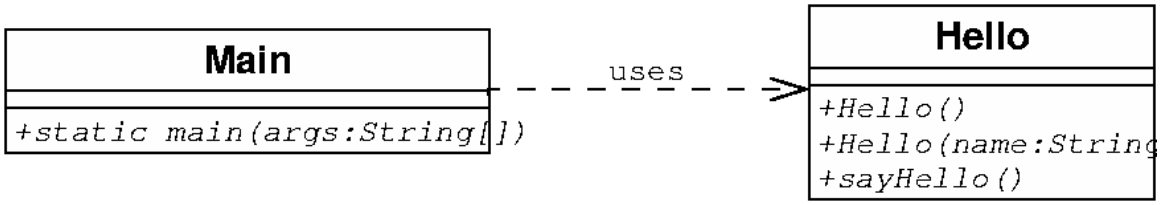
The program that we will develop is a kind of "helloworld" example. We will increase the complexity of the example, so that you familiarize yourself with different features of ProActive.

- First, we will code a "client-server" application, the server being an active object.
- Second, we will see how we can control the activity of an active object.
- Third, we will add mobility to this active object.
- Eventually, we will attach a graphical interface to the active object, and we will show how to move the widget between virtual machines (like in the penguin example).

29.1. The client - server example

This example implements a very simple client-server application. You will find it here [../HelloWorld.xml], as it is the HelloWorld example earlier mentionned in this manual. A client object displays a `String` gotten from a remote server.

The corresponding class diagram is the following :



29.2. Initialization of the activity

Active objects, as indicates their name, have an activity of their own (an internal thread).

It is possible to add pre and post processing to this activity, just by implementing the interfaces InitActive and EndActive, that define the methods initActivity and endActivity.

The following example will help you to understand how and when you can initialize and clean the activity.

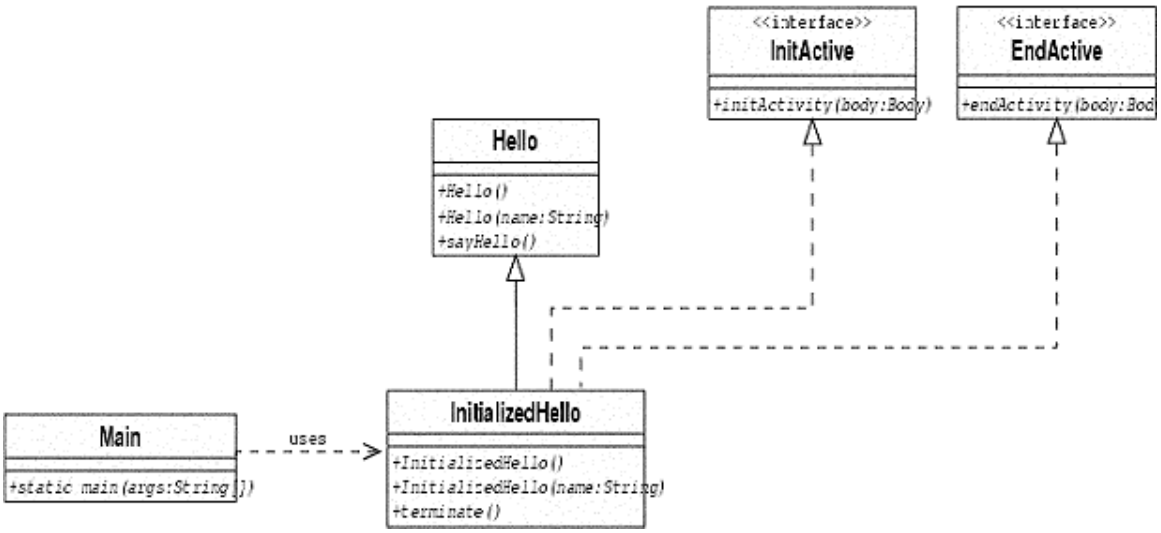
When instanciated, the activity of an object is automatically started, but it will first do what is written in the initActivity method.

Ending the activity can only be done from inside the active object (i.e. from a call to its own body). This is the reason why we have written a terminate method in the following example.

29.2.1. Design of the application

The InitializedHello class extends the Hello class, and implements the interfaces InitActive and EndActive.It acts a a server for the InitializedHelloClient class.

The main method is overriden so that it can instantiate the InitializedHello class



-->

29.2.2. Programming

29.2.2.1. InitializedHello

The source code of the InitializedHello class is here [hands_on_programming/code/InitializedHello.java.xml].

initActivity and endActivity here just log messages onto the console, so you can see when they are called.

initActivity is called at the creation of the active object, while endActive is called after the activity has terminated (thanks to the method terminate).

Here is the initActivity method :

```
public void initActivity(Body body) {
      System.out.println("I am about to start my activity");
}
```

Here is the endActivity method :

```
public void endActivity(Body body) {
      System.out.println("I have finished my activity");
}
```

The following code shows how to terminate the activity of the active object :

```
public void terminate() {
      // the termination of the activity is done through a call on the
      // terminate method of the body associated to the current active ob\
ject
      ProActive.getBodyOnThis().terminate();
}
```

The only differences from the the previous example is the classes instantiated, which are now InitializedHello (and not Hello) and InitializedHelloClient, and you will add a call to hello.terminate().

InitializedHello: Code is here [hands_on_programming/code/InitializedHello.java.xml].

InitializedHelloClient: Code is here [hands_on_programming/code/InitializedHelloClient.java.xml].

So, create InitializedHelloClient.java and InitializedHello.java in src/org/objectweb/proactive/examples/hello

Now compile all proactive sources

```
cd compile
windows>build.bat examples
linux>build examples
cd ..
```

Add "./classes" directory to CLASSPATH to use these two new source files

```
windows>set CLASSPATH=.;.\classes;.\ProActive_examples.jar;.\ProActive.jar;.\lib\bcel.ja
CLASSPATH=.:./classes:./ProActive_examples.jar:./ProActive.jar:./lib/bcel.jar:./lib/asm.
```

29.2.3. Execution

Execution is similar to the previous example; just use the InitializedHelloClient client class and InitializedHello server class.

29.2.3.1. Starting the server

```
linux> java -Djava.security.policy=scripts/proactive.java.policy -Dlog4j.configuration=f
org.objectweb.proactive.examples.hello.InitializedHello windows> java
-Djava.security.policy=scripts\proactive.java.policy -Dlog4j.configuration=file:scripts\
org.objectweb.proactive.examples.hello.InitializedHello &
```

29.2.3.2. Launching the client

```
linux> java -Djava.security.policy=scripts/proactive.java.policy -Dlog4j.configuration=f
org.objectweb.proactive.examples.hello.InitializedHelloClient //localhost/Hello
windows> java -Djava.security.policy=scripts\proactive.java.policy
-Dlog4j.configuration=file:scripts\proactive-log4j org.objectweb.proactive.examples.hell
//localhost/Hello
```

29.3. A simple migration example

This program is a very simple one : it creates an active object that migrates between virtual machines. It is a extension of the previous client-server example, the server now being mobile.

29.3.1. Required conditions

The conditions for MigratableHello to be a migratable active object are :

- it must have a constructor without parameters : this is a result of a ProActive restriction : the active object having to implement a no-arg constructor. </ p>

- implement the Serializable interface (as it will be transferred through the network).</>

Hello, the superclass, must be able to be serialized, in order to be transferred remotely. It does not have to implement directly java.io.Serializable, but its attributes should be serializable - or transient. For more information on this topic, check the manual [http://www-sop.inria.fr/oasis/ProActive/doc/api/org/objectweb/proactive/doc-files/Migration.xml].

29.3.2. Design

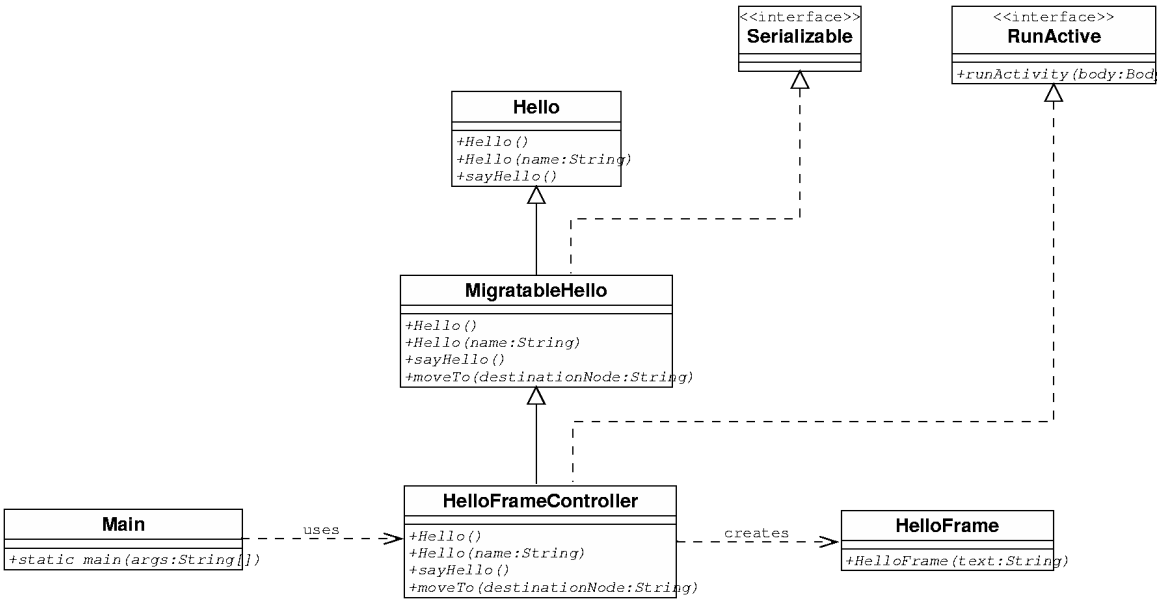
We want to further enhance InitializedHello it by making migratable : we'd like to be able to move it across virtual machines.

Thus, we create a MigratableHello class, that derives from InitializedHello. This class will implement all the non-functionnal behavior concerning the migration, for which this example is created. The Hello class (and InitializedHello) is left unmodified.

Note that the migration has to be initiated by the active object itself. This explains why we have to write the moveTo method in the code of MigratableHello - i.e. a method that contains an explicit call to the migration primitive. (cf migration documentation [http://www-sop.inria.fr/oasis/ProActive/doc/api/org/objectweb/proactive/doc-files/Migration.xml])

MigratableHello also implements a factory method for instanciating itself as an active object : `static MigratableHello createMigratableHello(String : name)`

The class diagram for the application is the following :



29.3.3. Programming

29.3.3.1. a) the MigratableHello class

The code of the MigratableHello class is here [hands_on_programming/code/MigratableHello.java.xml].

MigratableHello derives from the Hello class from the previous example

MigratableHello being the active object itself, it has to :

- implement the Serializable interface

- provide a no-arg constructor

- provide an implementation for using ProActive's migration mechanism.

A new method getCurrentNodeLocation is added for the object to tell the node where it resides..

A factory static method is added for ease of creation.

The migration is initiated by the moveTo method :

```
/** method for migrating
* @param destination_node destination node
*/
public void moveTo(String destination_node) {
      System.out.println("\n-----------------------------");
      System.out.println("starting migration to node : " + destination_node\
);
      System.out.println("...");
      try {
            // THIS MUST BE THE LAST CALL OF THE METHOD
            ProActive.migrateTo(destination_node);
      } catch (MigrationException me) {
            System.out.println("migration failed : " + me.toString());
      }
}
```

Note that the call to the ProActive primitive `migrateTo` is the last one of the method moveTo. See the manual [../Migration.xml] for more information.

29.3.3.2. c) the client class

The entry point of the program is written in a separate class : MigratableHelloClient [hands_on_programming/code/MigratableHelloC

It takes as arguments the locations of the nodes the object will be migrated to.

The program calls the factory method of MigratableHello to create an instance of an active object. It then moves it from node to node, pausing for a while between the transfers.

29.3.4. Execution

- start several nodes using the `startnode` script.

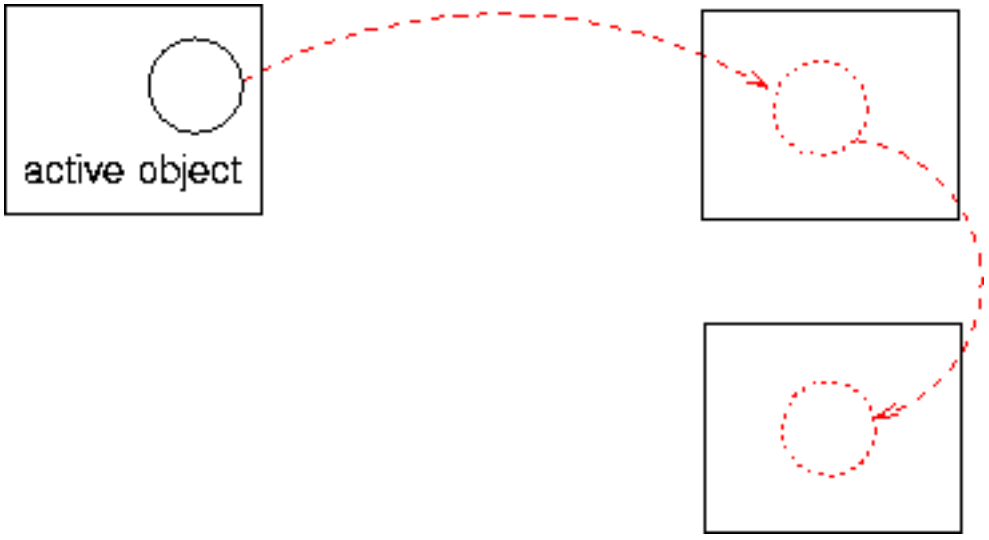
```
windows>cd scripts/windows
                        startNode.bat //localhost/n1
                        startNode.bat //localhost/n2
linux>cd scripts/linux
            ./startNode.sh //localhost/n1
            ./startNode.sh //localhost/n2
```

- compile and run the program (run MigratableHelloClient), passing in parameter the urls of the nodes you'd like the agent to migrate to.

```
cd compile
windows>build.bat examples
linux>build examples
```

```
cd ..


linux>java -Djava.security.policy=scripts/proactive.java.policy -Dlog4j.configuration=fi
org.objectweb.proactive.examples.hello.MigratableHelloClient //localhost/n1
//localhost/n2


windows>java -Djava.security.policy=scripts\proactive.java.policy
-Dlog4j.configuration=file:scripts\proactive-log4j org.objectweb.proactive.examples.hell
//localhost/n1 //localhost/n2
```

- observe the instance of MigratableHello migrating :



During the execution, a default node is first created. It then hosts the created active object. Then the active object is migrated from node to node, each time returning "hello" and telling the client program where it is located.

29.4. migration of graphical interfaces

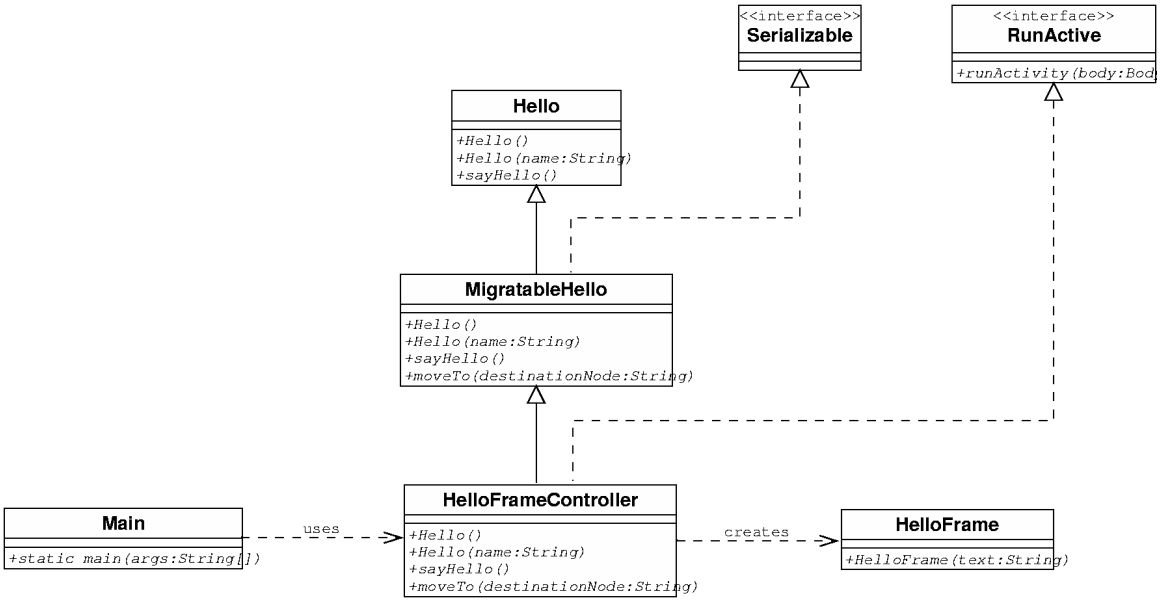
Graphical interfaces are not serializable, yet it is possible to migrate them with ProActive.

The idea is to associate the graphical object to an active object. The active object will control the activation and desactivation of this graphical entity during migrations.

Of course, this is a very basic example, but you can later build more sophisticated frames.

29.4.1. Design of the application

We will write a new active object class, that extends MigratableHello. The sayHello method will create a window containing the hello message. This window is defined in the class HelloFrame



29.4.2. Programming

29.4.2.1. HelloFrameController

The code of the HelloFrameController is here [hands_on_programming/code/HelloFrameController.java.xml].

This class extends MigratableHello, and adds an activity and a migration strategy manager to the object .

It creates a graphical frame upon call of the sayHello method.

Here we have a more complex migration process than with the previous example. We need to make the graphical window disappear before and reappear in a new location after the migration (in this example though, we wait for a call to sayHello). The migration of the frame is actually controlled by a MigrationStrategyManager, that will be attached to the body of the active object.. An ideal location for this operation is the initActivity method (from InitActive interface), that we override :

```
/**
 * This method attaches a migration strategy manager to the current active \
object.
 * The migration strategy manager will help to define which actions to take\
 before
 * and after migrating
 */
public void initActivity(Body body) {
      // add a migration strategy manager on the current active object
      migrationStrategyManager = new MigrationStrategyManagerImpl((Migrat\
able) ProActive.getBodyOnThis());
      // specify what to do when the active object is about to migrate
      // the specified method is then invoked by reflection
      migrationStrategyManager.onDeparture("clean");
}
```

The MigrationStrategyManager defines methods such as "onDeparture", that can be configured in the application. For example here, the method "clean" will be called before the migration, conveniently killing the frame :

```
public void clean() {
      System.out.println("killing frame");
      helloFrame.dispose();
      helloFrame = null;
      System.out.println("frame is killed");
}
```

29.4.2.2. HelloFrame

This is an example of a graphical class that could be associated with the active object. Here [hands_on_programming/code/HelloFrame.java.xml] is the code.

29.4.3. Execution

- Create a new class HelloFrameControllerClient: take the code of MigratableHelloClient used in the previous part, change the class declaration to HelloFrameControllerClient and replace the line

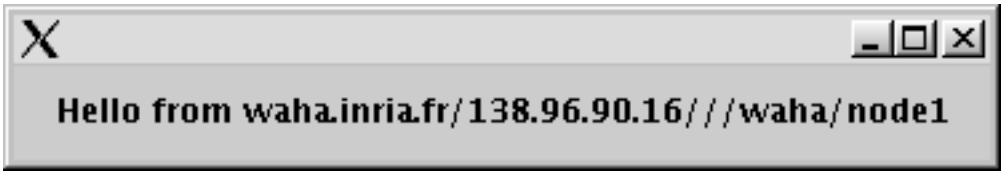
```
MigratableHello migratable_hello = MigratableHello.createMigratableHello("agent1");
```

with

```
MigratableHello migratable_hello = HelloFrameController.createHelloFrameController("ager
```

- Similarly to the simple migration example (use the HelloFrameControllerClient class), you will start remote nodes and specify a migration path.
- you have 2 ways for handling the display of the graphical objects :
- look on the display screens of the machines
- export the displays : in startNode.sh, you should add the following lines before the java command :

```
DISPLAY=myhost:0 export DISPLAY
```



The displayed window : it just contains a text label with the location of the active object.

Chapter 30. SPMD PROGRAMMING

30.1. OO SPMD on a Jacobi example

30.1.1. 1. Execution and first glance at the Jacobi code

30.1.1.1. 1.1 Source files: ProActive/src/org/objectweb/proactive/examples/jac

The Jacobi example is made of two Java classes:

- Jacobi.java: the main class

- SubMatrix.java: the class implementing the SPMD code

Have a first quick look at the code, especially the Jacobi class, looking for the strings "ProActive", "Nodes", "newSPMDGroup".

The last instruction of the class:

```
matrix.compute();
```

is an asynchronous group call. It sends a request to all active objects in the SPMD group, triggering computations in all the SubMatrix.

We will get to the class SubMatrix.java later on.

30.1.1.2. 1.2 Execution

ProActive examples come with scripts to easily launch the execution under both Unix and Windows.

For Jacobi, launch:

```
ProActive/scripts/unix/jacobi.sh
```

or

```
ProActive/scripts/windows/jacobi.bat
```

The computation stops after minimal difference is reached between two iterations (constant MINDIFF in class Jacobi.java), or after a fixed number of iteration (constant ITERATIONS in class Jacobi.java).

The provided script, using an XML descriptor, creates 4 JVMs on the current machine. The Jacobi class creates an SPMD group of 9 Active Objects; 2 or 3 AOs per JVM.

Look at the traces on the console upon starting the script; in the current case, remember that all JVMs and AOs send output to the same console. More specifically, understand the following:

```
- "Created a new registry on port 1099"
- "Reading deployment descriptor ...  Matrix.xml "
- "created VirtualNode"
```

```
- "**** Starting jvm on"
- "ClassFileServer is reading resources from classpath"
- "Detected an existing RMI Registry on port 1099""
- "Generating class :  ...  jacobi.Stub_SubMatrix "
- "ClassServer sent class ...  jacobi.Stub_SubMatrix successfully"
```

You can start IC2D (script ic2d.sh or ic2d.bat) in order to visualize the JVMs and the Active Objects. Just activate the "Monitoring a new host" in the "Monitoring" menu at the top left.

To stop the Jacobi computation and all the associated AOs, and JVMs, just ^C in the window where you started the Jacobi script.

30.1.2. 2. Modification and compilation

30.1.2.1. 2.1 Source modification

Do a simple source modification, for instance changing the values of the constants MINDIFF (0.00000001 for ex) and ITERATIONS in class Jacobi.java.

Caveat: Be careful, due to a shortcoming of the Java make system (ant), make sure to also touch the class SubMatrix.java that uses the constants.

30.1.2.2. 2.2 Compilation

ProActive distribution comes with scripts to easily recompile the provided examples:

```
linux>ProActive/compile/build
```

or

```
windows>ProActive/compile/build.bat
```

Several targets are provided (start build without arguments to obtain them). In order to recompile the Jacobi, just start the target that recompile all the examples:

```
build examples
```

2 source files must appear as being recompiled.

Following the recompilation, rerun the examples as explained in section 1.2 above, and observe the differences.

30.1.3. 3. Detailed understanding of the OO SPMD Jacobi

30.1.3.1. 3.1 Structure of the code

Within the class SubMatrix.java the following methods correspond to a standard Jacobi implementation, and are not specific to ProActive:

```
- internalCompute ()
```

```
- borderCompute ()
- exchange ()
- buildFakeBorder (int size)
- buildNorthBorder ()
- buildSouthBorder ()
- buildWestBorder ()
- buildEastBorder ()
- stop ()
```

The methods on which asynchronous remote method invocations take place

are:

```
- sendBordersToNeighbors ()
- setNorthBorder (double[] border)
- setSouthBorder (double[] border)
- setWestBorder (double[] border)
- setEastBorder (double[] border)
```

The first one sends to the appropriate neighbors the appropriate values, calling set*Border() methods asynchronously. Upon execution by the AO, the methods set*Border() memorize locally the values being received.

Notice that all those communication methods are made of purely functional Java code, without any code to the ProActive API.

On the contrary, the followings are ProActive related aspects:

```
- buildNeighborhood ()
- compute ()
- loop ()
```

We will detail them in the next section.

Note: the classes managing topologies are still under development. In the next release, the repetitive and tedious topology related instructions (e.g. methods buildNeighborhood) won’t have to be written explicitly by the user, whatever the topology (2D, 3D).

30.1.3.2. 3.2 OO SPMD behavior

Let us detail the OO SPMD techniques and ProActive related methods.

First of all, look for the definition and use of the attribute "asyncRefToMe". Using the primitive "getStubOnThis()", it provides a reference to the current active object **on which method calls are asynchronous**. It permits the AO to send requests to itself.

For instance in

```
this.asyncRefToMe.loop();
```

Notice the absence of classical loop. The method "loop()" is indeed asynchronously called from itself; it is not really recursivity since it does not have the drawback of the stack growing. It features an important advantage: the AO will remain reactive to other calls being sent to him. Moreover, it eases reuse since it is not necessary to explicitly encode within the main SPMD loop all the messages that have to be taken into account. It also facilitates composition since services can be called by activities outside the SPMD group, they will be automatically executed by the FIFO service of the Active Object.

The method "buildNeighborhood ()" is called only once for initialization. Using a 2D topology (Plan), it constructs references to north, south, west, east neighbors -- attributes with respective names. It also construct dynamically the group of neighbors. Starting from an empty group of type SubMatrix

```
this.neighbors = (SubMatrix) ProActiveGroup.newGroup

(SubMatrix.class.getName());
```

such typed view of the group is used to get the group view: Group neighborsGroup = ProActive-Group.getGroup(this.neighbors); Then, the appropriate neighbors are added dynamically in the group, e.g.:

```
neighborsGroup.add(this.north);
```

Again, the classes managing topologies will permit to simplify this code.

30.1.3.3. 3.3 Adding a Method barrier for a step by step execution

Let say we would like to control step by step the execution of the SPMD code. We will add a barrier in the SubMatrix.java, and control the barrier from input in the Jacobi.java class.

In class SubMatrix.java, add a Method `barrier()` of the form:

```
String[] st= new String[1];

st[0]="keepOnGoing";

ProSPMD.barrier(st);
```

Do not forget to define the `keepOnGoing()` method that indeed can return void, and just be empty. Find the appropriate place to call the `barrier()` Method in the `loop()` Method.

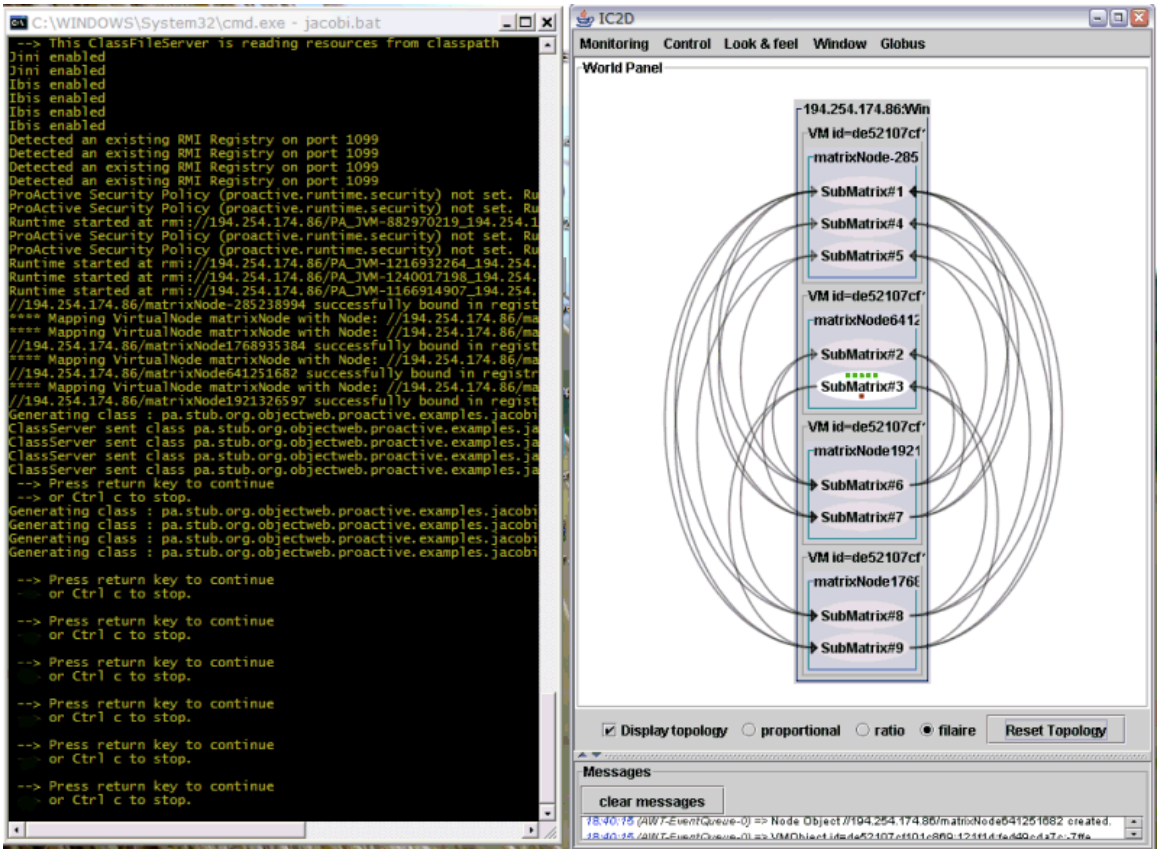
In class Jacobi.java, just after the `compute()` Method, add an infinite loop that, upon a user's return key pressed, calls the method `keepOnGoing()` on the SPMD group "matrix". Here are samples of the code:

```
while (true) {

printMessageAndWait();

matrix.keepOnGoing();

}

...

private static void printMessageAndWait() {

java.io.BufferedReader d = new java.io.BufferedReader(

new java.io.InputStreamReader(System.in));

System.out.println(" --> Press return key to continue");

System.out.println(" or Ctrl c to stop.");
```

```
try {

d.readLine();

} catch (Exception e) {

}

}
```

Recompile, and execute the code. Each iteration needs to be activated by hitting the return key in the shell window where Jacobi was launched. Start IC2D (./ic2d.sh or ic2d.bat), and visualize the communications as you control them. Use the "Reset Topology" button to clear communication arcs. The green and red dots indicate the pending requests.

You can imagine and test other modifications to the Jacobi code.



30.1.3.4. 3.4 Undestanding various different kind of barriers

The group of neighbors built above is important wrt synchronization. Below in method "loop()", an efficient barrier is achieved only using the direct neighbors:

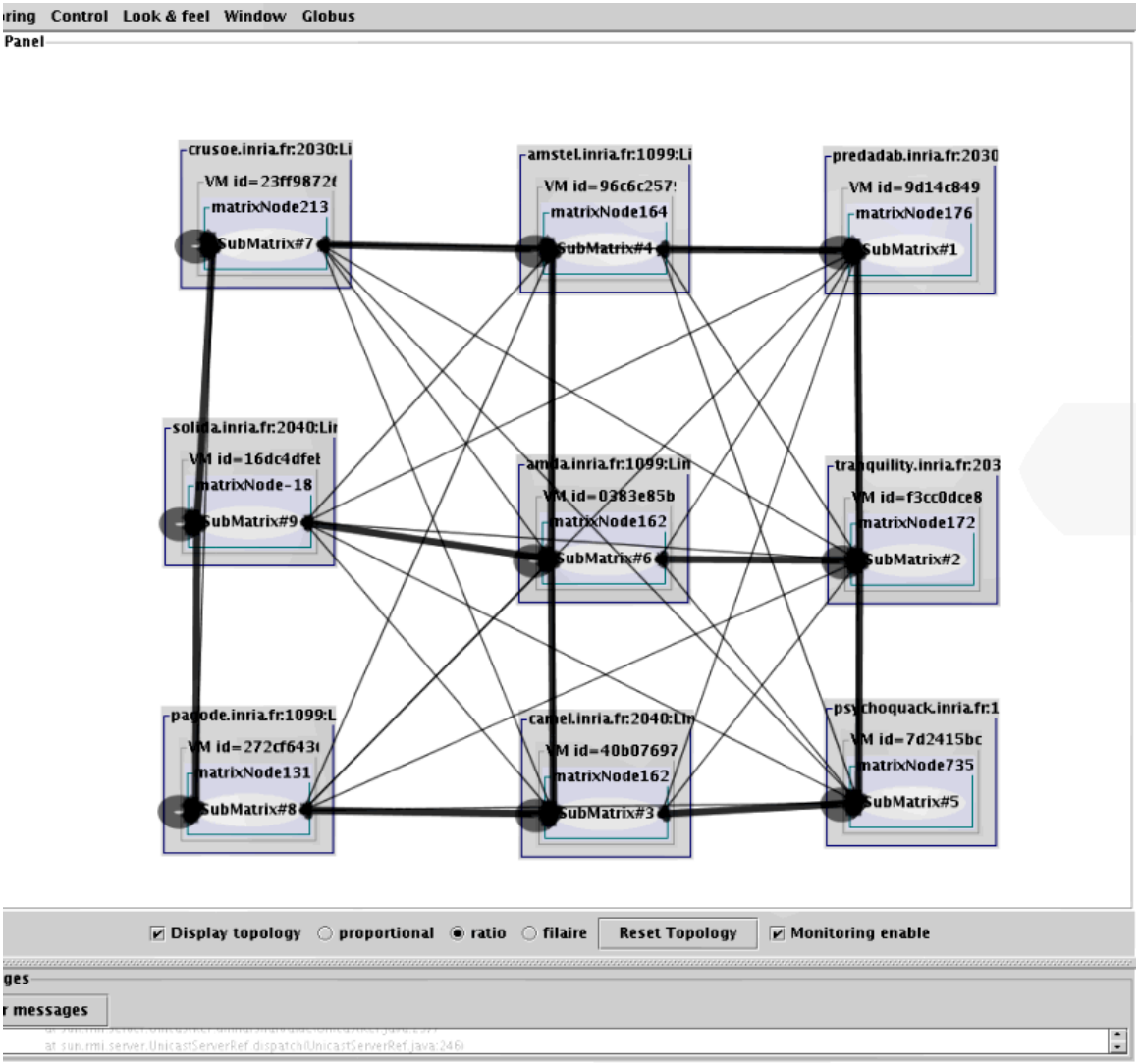
```
ProSPMD.barrier("SynchronizationWithNeighbors"+ this.iterationsToStop,
this.neighbors);
```

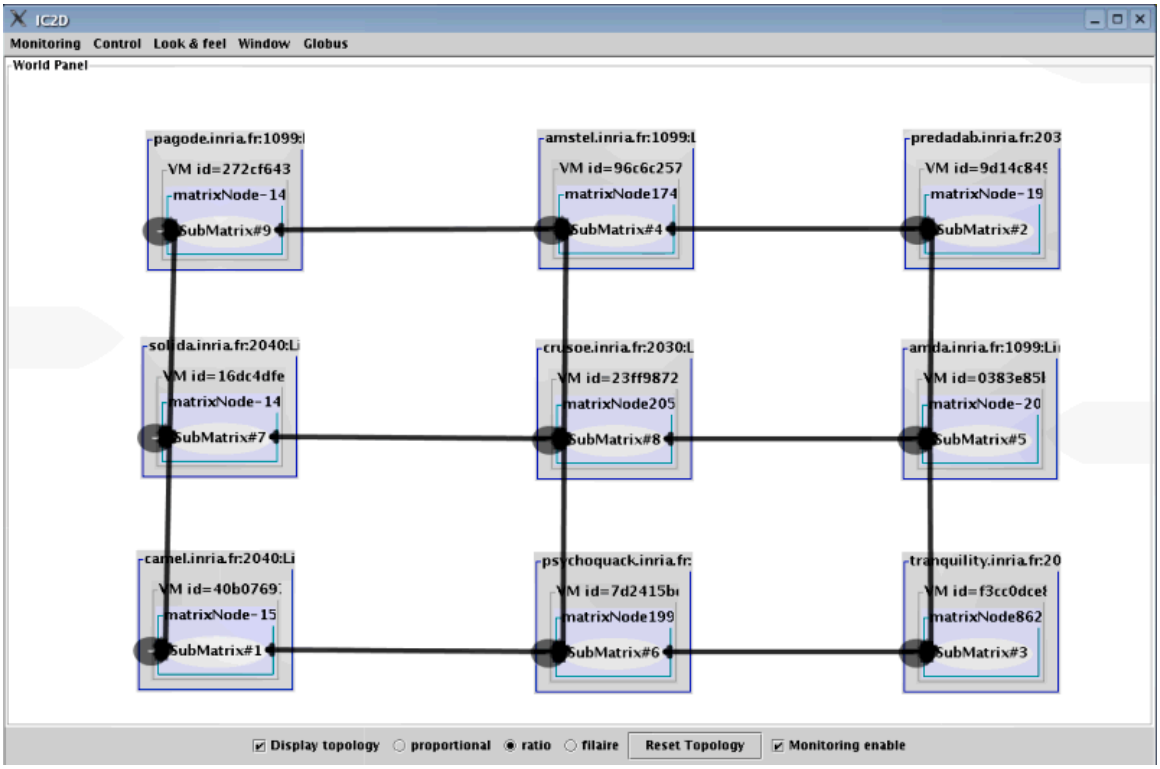
This barrier takes as a parameter the group to synchronize with: it will be passed only when the 4 neighbors in the current 2D example have reached the same point. Adding the rank of the current iteration allows to have a unique identifier for each instance of the barrier.

Try to change the barrier instruction to a total barrier:

```
ProSPMD.barrier("SynchronizationWithNeighbors"+ this.iterationsToStop);
```

Then recompile and execute again. Using IC2D observe that many more communications are necessary.





In order to get details and documentation on Groups and OO SPMD, have a look at:

`ProActive/src/org/objectweb/proactive/doc-files/`

`TypedGroupCommunication.html`

`OOSPMD.html`

30.1.4. 4. Virtual Nodes and Deployment descriptors

30.1.4.1. 4.1 Virtual Nodes

Get back to the source code of Jacobi.java, and understand where and how the Virtual Nodes and Nodes are being used.

30.1.4.2. 4.2 XML Descriptors

The XML descriptor being used is:

`ProActive/descriptors/Matrix.xml`

Look for and understand the following definitions:

`- Virtual Node Definition`

`- Mapping of Virtual Nodes to JVM`

`- JVM Definition`

`- Process Definition`

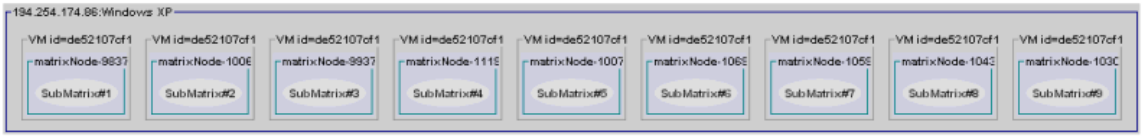
A detailed presentation of XML descriptors is available at:

`ProActive/docs/api/index.html`

entry 9. XML Deployment Descriptors

30.1.4.3. 4.3 Changing the descriptor

Edit the file Matrix.xml in order to change the number of JVMs being used. For instance, if your machine is powerful enough, start 9 JVMs, in order to have a single SubMatrix per JVM.



You do not need to recompile, just restart the execution. Use IC2D to visualize the differences in the configuration.

30.1.5. 5. Execution on several machines and Clusters

30.1.5.1. 5.1 Execution on several machines in the room

- **Explicit machine names**

`ProActive/examples/descriptors/Matrix.xml` is the XML deployment file used in this tutorial to start 4 jvms on the local machine. This behavior is achieved by referencing in the creation tag of **Jvm1, Jvm2, Jvm3, Jvm4** a **jvmProcess** named with the id **localProcess**. To summarize briefly at least one **jvmProcess** must be defined in an xml deployment file. When this process is referenced directly in the creation part of the jvm definition (like the example below), the jvm will be created locally. On the other hand, if this process is referenced by another process(**rshProcess** for instance, this is the case in the next example), the jvm will be created remotely using the related protocol (rsh in the next example).

Note that several **jvmProcess** can be defined, for instance in order to specify different jvm configurations (e.g classpath, java path,...).

```
 <ProActiveDescriptor
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
  <componentDefinition>
   <virtualNodesDefinition>
    <virtualNode name="matrixNode"
 property="multiple"/>
   </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
   <mapping>
    </map>
    <map virtualNode="matrixNode">
     <jvmSet>
      <vmName value="Jvm1"/>
      <vmName value="Jvm2"/>
      <vmName value="Jvm3"/>
      <vmName value="Jvm4"/>
     </jvmSet>
    </map>
   </mapping>
   <jvms>
    <jvm name="Jvm1">
```

```
    <creation>
     <processReference refid="localProcess"/>
    </creation>
   </jvm>
   <jvm name="Jvm2">
    <creation>
     <processReference refid="localProcess"/>
    </creation>
   </jvm>
   <jvm name="Jvm3">
    <creation>
     <processReference refid="localProcess"/>
    </creation>
   </jvm>
   <jvm name="Jvm4">
    <creation>
     <processReference refid="localProcess"/>
    </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="localProcess">
    <jvmProcess
    class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

Modify your XML deployment file to use the current jvm (i.e the jvm reading the descriptor) and also to start 4 jvms on remote machines using **rsh protocol**.

Use IC2D to visualize the machines and the JVMs being launched on them.

```
<ProActiveDescriptor
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"↩
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="matrixNode"
property="multiple"/>
  </virtualNodesDefinition>
 </componentDefinition>
 <deployment>
  <mapping>
   </map>
   <map virtualNode="matrixNode">
    <jvmSet>
     <currentJvm />
     <vmName value="Jvm1"/>
     <vmName value="Jvm2"/>
     <vmName value="Jvm3"/>
     <vmName value="Jvm4"/>
    </jvmSet>
   </map>
```

```
  </mapping>
  <jvms>
   <jvm name="Jvm1">
    <creation>
     <processReference  refid="rsh_titi"/>
    </creation>
   </jvm>
   <jvm name="Jvm2">
    <creation>
     <processReference refid="rsh_toto"/>
    </creation>
   </jvm>
   <jvm name="Jvm3">
    <creation>
     <processReference
refid="rsh_tata"/>
    </creation>
   </jvm>
   <jvm name="Jvm4">
    <creation>
     <processReference
refid="rsh_tutu"/>
    </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="localProcess">
    <jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
   <processDefinition id="rsh_titi">
    <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="titi">
     <processReference
refid="localProcess"/>
    /rshProcess>
   </processDefinition>
   <processDefinition id="rsh_toto">
    <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="toto">
     <processReference
refid="localProcess"/>
    /rshProcess>
   </processDefinition>
   <processDefinition id="rsh_tata">
    <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="tata">
     <processReference
refid="localProcess"/>
    /rshProcess>
   </processDefinition>
   <processDefinition id="rsh_tutu">
    <rshProcess
```

```
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="tutu">
     <processReference  refid="localProcess"/>
    /rshProcess>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

Pay attention of what happened to your previous XML deployment file. First of all to use the current jvm the following line was added just under the **jvmSet** tag

```
<jvmSet>
 <currentJvm />
 ...
<jvmSet>
```

Then the jvms are not created directly using the localProcess, but instead using other processes named **rsh_titi, rsh_toto, rsh_tata, rsh_tutu**

```
<jvms>
 <jvm name="Jvm1">
  <creation>
   <processReference refid="rsh_titi"/>
  </creation>
 </jvm>
 <jvm name="Jvm2">
  <creation>
   <processReference refid="rsh_toto"/>
  </creation>
 </jvm>
 <jvm name="Jvm3">
  <creation>
   <processReference refid="rsh_tata"/>
  </creation>
 </jvm>
 <jvm name="Jvm4">
  <creation>
   <processReference refid="rsh_tutu"/>
  </creation>
 </jvm>
</jvms>
```

Those processes as shown below are rsh processes. Note that it is **mandatory** for such processes to reference a **jvmProcess**, in this case named with the id **localProcess**, to create, at deployment time, a jvm on machines titi, toto, tata, tutu, once connected to those machines with rsh.

```
<processDefinition id="localProcess">
 <jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
</processDefinition>
<processDefinition id="rsh_titi">
```

```
 <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="titi">
  <processReference refid="localProcess"/>
 /rshProcess>
</processDefinition>
<processDefinition id="rsh_toto">
 <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="toto">
  <processReference refid="localProcess"/>
 /rshProcess>
</processDefinition>
<processDefinition id="rsh_tata">
 <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="tata">
  <processReference refid="localProcess"/>
 /rshProcess>
</processDefinition>
<processDefinition id="rsh_tutu">
 <rshProcess
class="org.objectweb.proactive.core.process.rsh.RSHProcess"
hostname="tutu">
  <processReference refid="localProcess"/>
 /rshProcess>
</processDefinition>
```

- **Using Lists of Processes**
- You can also use the notion of **Process List**, which leads to the same result but often simplifies the xml. Two tags are provided:
  - **processListbyHost**

Used to list the machine's name in a single process

```
<ProActiveDescriptor
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="matrixNode"
property="multiple"/>
  </virtualNodesDefinition>
 </componentDefinition>
 <deployment>
  <mapping>
   </map>
   <map virtualNode="matrixNode">
    <jvmSet>
     <currentJvm/>
     <vmName value="Jvm1"/>
    </jvmSet>
   </map>
  </mapping>
```

```
  <jvms>
   <jvm name="Jvm1">
    <creation>
     <processReference
refid="rsh_list_titi_toto_tutu_tata"/>
    </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition
id="localProcess">
    <jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
   <processDefinition
id="rsh_list_titi_toto_tutu_tata">
    <processListbyHost
class="org.objectweb.proactive.core.process.rsh.RSHProcessList"
hostlist="titi toto tata tutu">
     <processReference
refid="localProcess"/>
    </processListbyHost>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

- **processList**

Used when machine's names follow a list format, for instance titi1 titi2 titi3 ... titi100

```
          <ProActiveDescriptor
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="matrixNode"
property="multiple"/>
  </virtualNodesDefinition>
 </componentDefinition>
 <deployment>
  <mapping>
   </map>
   <map virtualNode="matrixNode">
    <jvmSet>
     <currentJvm/>
     <vmName value="Jvm1"/>
    </jvmSet>
   </map>
  </mapping>
  <jvms>
   <jvm name="Jvm1">
    <creation>
     <processReference
```

```
refid="rsh_list_titi1_to_100"/>
    </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition
id="localProcess">
    <jvmProcess
class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
   <processDefinition
id="rsh_list_titi1_to_100">
    <processList
class="org.objectweb.proactive.core.process.rsh.RSHProcessList"
fixedName="titi" list="[1-100]"
domain="titi_domain">
     <processReference
refid="localProcess"/>
    </processList>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

30.1.5.2. 5.2 Execution on Clusters

If you have access to your own clusters, configure the XML descriptor to launch the Jacobi on them, using the appropriate protocol:

ssh, LSF, PBS, Globus, etc.

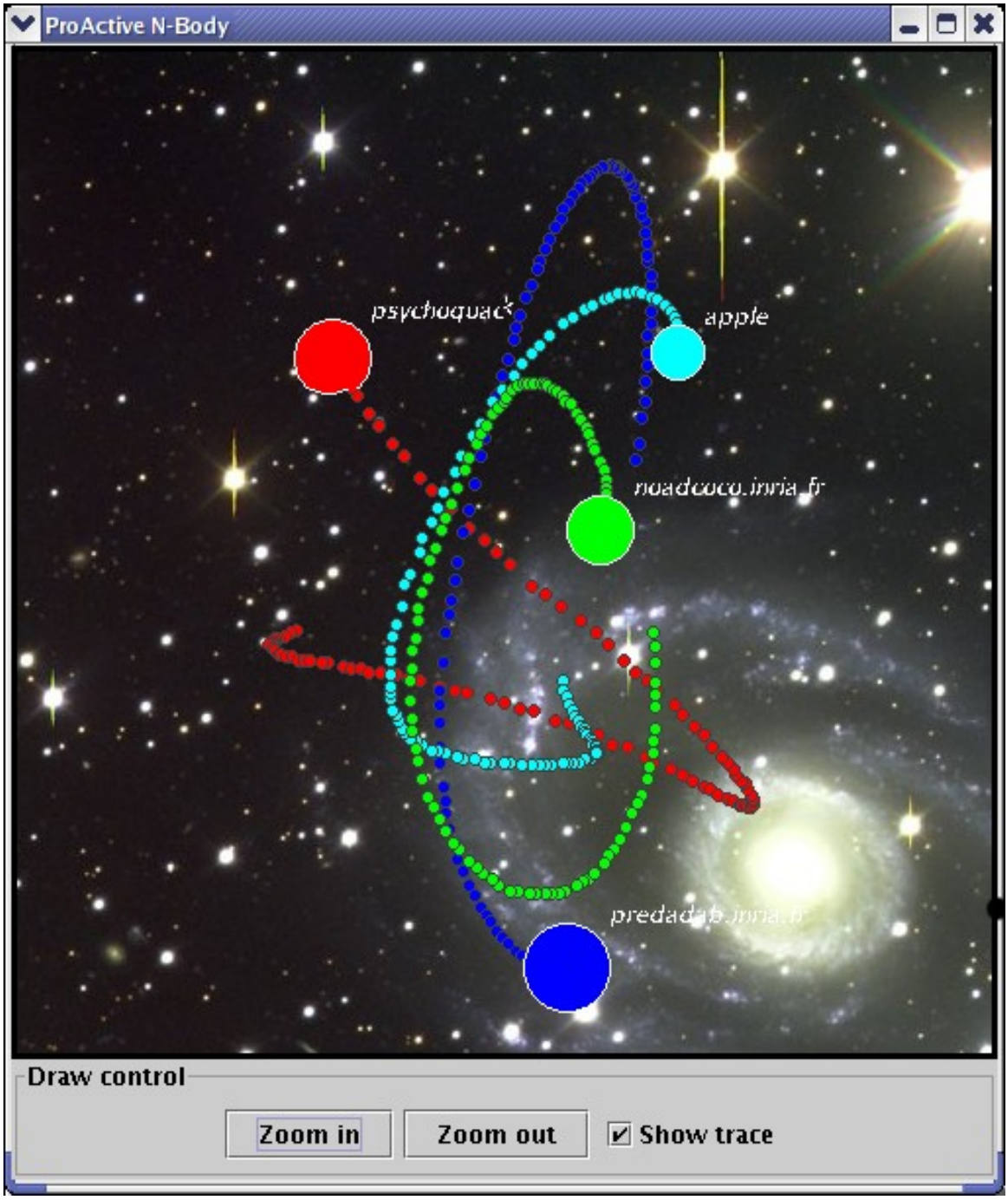
Have a look at XML Descriptors documentation [http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/Descriptor.xml] to get the format of the XML descriptor for each of the supported protocols.

Chapter 31. 5. The nbody example

31.1. Using facilities provided by ProActive on a complete example

31.1.1. 1 Rationale and overview

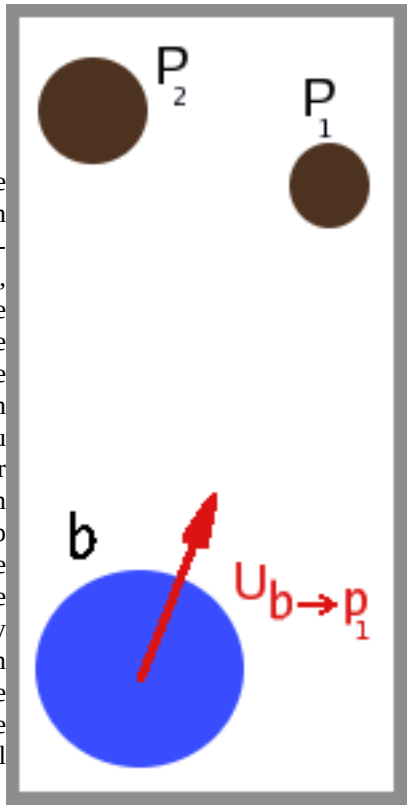
This section of the guided tour goes through the different steps that could take you to writing an application with ProActive, from a simple design, to a more complicated structure. This is meant to help you get familiar with the Group facilities offered by ProActive. Please take note that this page tries to take you through the progression, step by step. You may find some more information [http://www-sop.inria.fr/oasis/proactive/apps/nbody.xml], mainly on the design, on the web page of the applications/examples [http://www-sop.inria.fr/oasis/proactive/apps/] of proactive. This is a snapshot of the ProActive nbody example running on 3 hosts with 8 bodies:



n-body is a classic problem. It consists in working out the position of bodies in space, which depend only on the gravitational forces that apply to them. A good introduction to the problem is given here [http://www.cs.berkeley.edu/%7Esouravc/cs267/nbody.htm]. You may find a detailled explanation of the underlying mathematics here [http://members.fortunecity.com/kokhuitan/nbody.xml]. Different ways of finding numerical solutions are given here [http://www.amara.com/papers/nbody.xml].

In short, one considers several bodies (sometimes called particles) in space, where the only force is due to gravity. When only two bodies are at hand, this is expressed as

$$F_{p \to b} =$$



$F_{p->b}$ is the force that p applies on b, G is the gravitational constant, $m_p m_b$ describe the mass of the bodies, r is the distance between p and b, andu is a unit vector in the direction going from p to b. When we consider all the forces that apply to one given body, we have to sum up the contribution of all the other bodies :

$$F_b = \sum_{p \in Planets} F_{p \to b}$$

This should be read as : the total force on the body b is the sum of all the forces applied to b, generated by all the other bodies in the system.

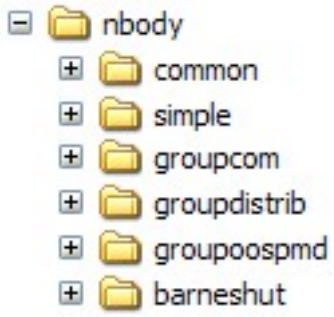
This is the force that has to be computed for every body in the system. With this force, using the usual physics formulae, (Newton's second Law)

$$F_b = ma$$

one may now

31.1.2. 2 Source files: ProActive/src/org/objectweb/proactive/exa

This guided tour is based on the files you may find in the directory ProActive/src/org/objectweb/proactive/examples/nbody. You'll find the following tree:



The common directory contains files reused through the different version. 'simple' is the simplest example, groupcom is the first example with Group communication, and 'groupdistrib' and 'groupoospmd' are two enhancement based on different synchronization schemes. 'barneshut' is a bit special, in that it contains a different algorithm to solve the nbody problem.

31.1.3. 3 Common files

The files contained in 'common' are those that are reused throughout the different versions. Let's see what they do:

- First of all there are the two files called Displayer.java and NBodyFrame.java. These handle the graphical output of the bodies, as they move about in space. They are not specially of interest, as GUI is not the point of this tutorial. Nonetheless, please note that the important method here is

  ```
  public void drawBody(int x, int y, int vx, int vy, int weight, int d,
  int id) ;
  ```

  Taking position, velocity, diameter and unique identifier of a body, it updates the display window.

- Then, we have the files Force.java and Planet.java. They are used to compute the interaction between two distant bodies in the universe. Since they are in the common directory, they can be modified to include other forces (for example, collision) in a simple manner, which would be spread to all the examples. A Planet is no more than a point in space, with velocity and mass - the diameter expresses the size to use for the display:

  ```
  public class Planet implements Serializable{
     public double mass;
     public double x,y,vx,vy;
        // position and velocity
     public double diameter;
          // diameter of the body, used by the Displayer
     ...
  ```

  Please take note that it implements Serializable because it will be sent as parameter to method calls on Active Objects, but it is good practice to have all your ProActive classes implement Serializable. For example, migration requires everything to implement it, fault-tolerance also....

  The Force class is just the implementation of what a physical force really is. It is the implementation of a 2D vector, with the method add following the physics rule

$$F_{p \to b} = \frac{-G m_p m_b}{r^2} \overrightarrow{u}_{p \to b}$$

- Point2D.java and Rectangle.java are helper files. They simply implement what a point in space looks like, and what a rectangle is. They were created to avoid the trouble of using the standard java java.awt.geom.Point2D and java.awt.geom.Rectangle2D, which carry a bit too much overhead, and also to have them Serializable.

- And finally, the Start.java acts as the wrapper for the main() method. There is a part which reads command line parameters, counting bodies and iterations, and constructing the optional Displayer. Before choosing which example to run, it creates the nodes required by the simulation :

```
        // Construct deployment-related variables: pad & nodes
        descriptorPad = null;
        VirtualNode vnode;
        try { descriptorPad = ProActive.getProactiveDescriptor(xmlFileName);↩
}
        catch (ProActiveException e) { abort(e); }
        descriptorPad.activateMappings();
        vnode = descriptorPad.getVirtualNode("Workers");
        Node[] nodes = null;
        try { nodes = vnode.getNodes(); }
        catch (NodeException e) { abort(e);
 }
```

the `Node [] nodes` are the different JVMs that were created on possibly different machines. They are used for Active Object creation. They were specified in the descriptor used to deploy the application. You may find more information on these descriptors here [http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/Descriptor.xml], while Active Object creation is explained in this page [http://www-sop.inria.fr/oasis/proactive/doc/api/org/objectweb/proactive/doc-files/ActiveObjectCreation.xml]. Just as an example, in the simple package, the Maestro is created on the first of theses JVMs, and takes three parameters, a `Domain []`, an `Integer`, and a `Start` (it will be detailed later) :

```
        Object [] constructorParams ;
        constructorParams = {domainArray, new Integer(maxIter), killsupport}↩
;
        maestro = (Maestro) ProActive.newActive
          ( Maestro.class.getName(), constructorParams , nodes[0] ) ;
```

The files contained in the other directories, 'simple', 'groupcom', 'groupdistrib' , 'groupoospmd' , detail steps of increasing complexity, making the application use different concepts. 'barneshut' contains the final implementation, featuring the Barnes-Hut algorithm. But let's not go too fast. Let's have a look at the insides of the simplest implementation of the n-body problem.

31.1.4. 4 Simple Active Objects

This is the implementation of the simplest example of nbody. We defined the `Planet` to be a passive object, and it does nothing. It is a container for position, velocity and mass, as we've seen in the description given higher up. The real actors are the `Domains`, they do all the work. To every `Planet` in the universe, is associated a `Domain`, which is an `Active Object`. This `Domain` contains the code to manage the communication of the possitions of the `Planets` during the simulation. They are created in the Start.java file :

```
    Rectangle universe = new Rectangle (-100,-100,100,100);
```

```
Domain [] domainArray = new Domain [totalNbBodies];
for (int  i = 0 ; i < totalNbBodies ; i++)  {
   Object [] constructorParams = new Object [] {
          new Integer(i),
          new Planet (universe)
      };
   try {
      // Create all the Domains used in the simulation
      domainArray[i] = (Domain) ProActive.newActive(
            Domain.class.getName(),
            constructorParams,
            nodes[(i+1) % nodes.length]
      );
   }
   catch (ActiveObjectCreationException e) { killsupport.abort(e); }
   catch (NodeException e) {  killsupport.abort(e); }
}
```

See how the call to `ProActive.newActive` creates one new `Active Object`, a `Domain`, at each iteration of the loop. The array `nodes` contains all the nodes on which an Active Object may be deployed; at each iteration, one given node, ie one JVM, is selected. The `constructorParams` are the parameters that are to be passed to the constructor of `Domain`, and since it's an `Object []` , the parameters may only be `Objects` (don't try to build constructors using ints in their constructor - this explains the use of the class `Integer`).

The `Domains`, once created, are initialized, and then they synchronize themselves by all pinging the maestro, with the notifyFinished call:

```
// init workers, from the Start class
for (int i=0 ; i < totalNbBodies ; i ++)
   domainArray[i].init(domainArray, displayer, maestro);
 // init method, defined within each worker



 public void init(Domain [] domainArray, Displayer dp, Maestro master) {
       this.neighbours = domainArray;
       .....
       maestro.notifyFinished();  // say we're ready to start
  }



public void notifyFinished() {
   this.nbFinished ++;
   if (this.nbFinished == this.domainArray.length) {
      this.iter ++;
      if (this.iter==this.maxIter)
         this.killsupport.quit();
      this.nbFinished = 0 ;
        for (int i= 0 ; i < domainArray.length ; i++)
           this.domainArray[i].sendValueToNeighbours();
  }
}
```

Notice how domainArray is passed to all the `Domains`, when calling `init`. This is the value assigned to the local field `neighbours`, which later on serves to communicate with all the other `Domains` of the simulation.

The synchronization is done by the `Maestro`, which counts the number of `Domains` that have finished, and then asks them to go on to the next iteration. While in their execution, the `Domains` gather information concerning the position of all the other bodies, which need to be known to move the local `Planet`, at every time step. This is done using a push schema : instead of explicitly asking for information, this information is automatically issued :

```
     public void sendValueToNeighbours() {
         for (int i = 0 ; i < this.neighbours.length ; i ++)
           if (i != this.identification) // don't notify self!
              this.neighbours[i].setValue(this.info, this.identification);
          .....
         }

      public void setValue(Planet inf, int id) {
        this.values [id] = inf;
        this.nbReceived ++ ;
        if (this.nbReceived > this.nbvalues)  // This is a bad sign!
           System.err.println("Domain " + identification + " received too↩
many answers");
        if (this.nbReceived == this.nbvalues) {
           this.maestro.notifyFinished();
           moveBody();
        }
     }
```

This means that each `Domain` sends its information to all the other `Domains`, and then waits until it has received all the positions it is waiting for. The other `Domains` are stored as an array, which is called `neighbours`. You may find another view of this example on this web page [http://www-sop.inria.fr/oasis/proactive/apps/nbody-simple.xml].

31.1.5. 5 Groups of Active objects

This is a simple improvement, which allows to have faster communication. You may have noticed the Group capabilities of ProActive. They give us the ability to call an operation on an object which is a Group, and have it sent to all the members of the Group. We can use them in this framework : first, create a Group (instead of having independant `Active Objects`) :

```
        // in the Start class
         Object [][] params = ...
         Domain  domainGroup = null;
         try {
            // Create all the Domains as part of a Group
            domainGroup = (Domain) ProActiveGroup.newGroup (↩
Domain.class.getName(), params, nodes);
         }
         catch ....>
```

The double array params stores the parameters passed to the constructors of the `Domains` we're creating. `Domain 0` will have `params[0][]` passed as arguments, `Domain 1 params[1][]`, and so on. The nodes are the Nodes on which to create these Active Objects. Do notice the `try... catch` construction, which is needed, like around any creation of `Active Objects`, because it may raise exceptions. In this previous bit of code, a `Group` containing new `Active Objects` has been created, and all these Objects belong to the group . You may have noticed that the type of the `Group` is Domain. It's a bit strange at first, and you may think this reference points to only one Active Object at once, but that's not true. We're accesssing all the objects in the

group, and to be able to continue using the methods of the Domain class, the group is **typed** as `Domain`, and that's the reason why it's called a **typed Group**.

Then, this group is passed as parameter to all the members of the Group, in just one call:

```
// Still in the Start class
  domainGroup.init(domainGroup, displayer, maestro);
```

This method sets the local field as a copy of the passed parameter, and as such is unique, and we can play around with it without affecting the others. So let's remove the local `Domain` from the Group, to avoid having calls on self:

```
          public void init(Domain domainGroup, Displayer dp, Maestro
master) {
            this.neighbours = domainGroup;
            Group g = ProActiveGroup.getGroup(neighbours);
            g.remove(ProActive.getStubOnThis()); // no need to send
information to self
          .....
```

Remember that in the previous example, the neighbours where stocked in an array, and each was accessed in turn:

```
for (int i = 0 ; i < this.neighbours.length ; i ++)
    if (i != this.identification) // don't notify self!
      this.neighbours[i].setValue(this.info, this.identification);
```

Well, that's BAAAAD, or at least inefficient! Replace this by the following code, because it works faster :

```
this.neighbours.setValue(this.info, this.identification);
```

This has the following meaning : call the method setValue, with the given parameters, on all the members of the Group `neighbours`. In one line of code, the method `setValue` is called on all the Active Objects in the group.

You may find another view of this example on this web page [http://www-sop.inria.fr/oasis/proactive/apps/nbody-groupcom.xml].

31.1.6. 6 groupdistrib

Now, do like the idea that the synchronization is centralized on one entity, the `Maestro`? I don't and it's the bottleneck of the application anyway : once a `Domain` has finished, it sends the notifyFinshed, and then sits idle. Well, a way of making this better is to remove this bottleneck completely! This is done by using an odd-even scheme : if a Domain receives information from a distant `Domain` too early (ie in the wrong iteration), this information is stored, and will get used at the next iteration. In the meantime, the local Domain does not change its iteration, because it is still waiting for more results, in the current iteration.

```
public void setValue(Planet inf, int receivedIter) {
  if (this.iter == receivedIter) {
    this.currentForce.add(info, inf);
    this.nbReceived ++ ;
    if (this.nbReceived == this.nbvalues)
       moveBody();
```

```
      }
      else {
         this.prematureValues.add(new Carrier (inf, receivedIter));
      }
   }
```

Also notice how the computation is done incrementally when the result is received (`this.currentForce.add(info, inf);`), instead of when all the results have arrived. This allows for less time spent idle. Indeed, waiting for all the results before computing might leave idle time between setValue requests. And then, just before computing the new position of the body, the sum of all the forces has to be computed. It's better to have this sum ready when needed.

The `prematureValues` Vector is the place where we put the values that arrive out of sync. When a value is early, it is queued there, and dequeued as soon as this `Domain` changes iteration.

```
      public void sendValueToNeighbours()  {
             reset();
             this.iter++;
             if (this.iter < this.maxIter) {
                neighbours.setValue(this.info, this.iter);
                ... // display related code
                treatPremature();
                }
              ... // JVM destruction related code
         }
```

That treatPremature() method simply treats the values that were early as if they had just arrived, by calling the setValue method with the parameters stored.

You may find another view of this example on this web page [http://www-sop.inria.fr/oasis/proactive/apps/nbody-groupdistrib.xml].

31.1.7. 7 Object Oriented SPMD Groups

This is another way to improve the groupcom example. It also removes the master, but this time by inserting oospmd barriers, that can be thought as behaving like the maestro class, but faster. To create functional OOspmd Groups, there is a special instruction, which takes the same parameters as a newGroup instruction :

```
      Object [][] params =  ...
      Domain domainGroup = null;
      try {
          domainGroup = (Domain) ProSPMD.newSPMDGroup(↩
Domain.class.getName(), params, nodes);
         }
         catch ...
```

Now, to use this OOspmd group properly, we want to use the barrier() methods. We put these in the Domains code, to do the synchronization. What happens is that each Domain hits the barrier call, and then waits for all the others to have reached it, before reading its request queue again.

```
public void sendValueToNeighbours() {
  this.neighbours.setValue(this.info, this.identification);
  ProSPMD.barrier("barrier" + this.iter);
```

```
  this.iter++;
  this.asyncRefToSelf.moveBody();
....
```

Beware, the stop-and-wait is not just after the barrier call, but instead blocks the request queue. So if there is code after that barrier, it will get executed. In fact, the barrier should be seen as a prioritary request on the queue. This explains why we had to put the code after the barrier as a method placed on an asynchronous refernce to self. If we hadn't done it that way, but just appended the code of that method just after the barrier, the call to `moveBody()` would be executed before the barrier execution, which is exactly what we don't want!

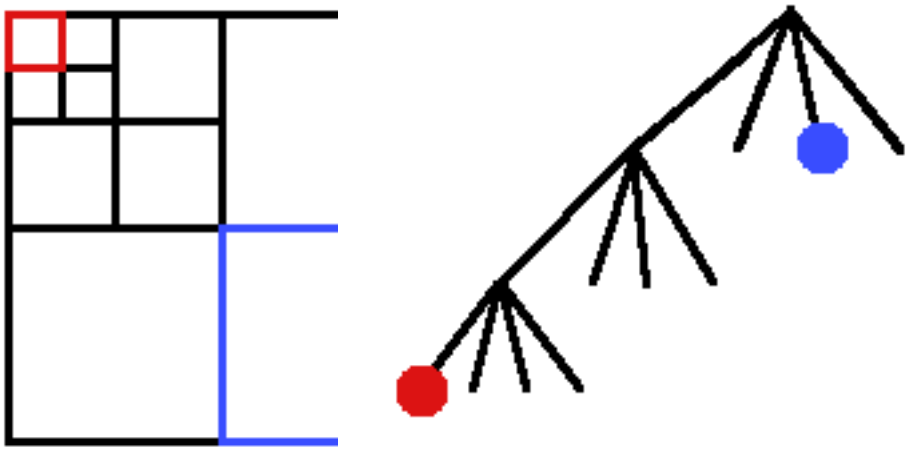
You may find another view of this example on this web page [http://www-sop.inria.fr/oasis/proactive/apps/nbody-groupoospmd.xml].

31.1.8. 8 Barnes-Hut

This way to construct the nbody simulation is based on a very different algorithm. This is inserted to show how one can express this algorithm in ProActive, but breaks off from the previous track, having such a different approach to solving the problem. Here's how it works:

To avoid broadcasting to every active object the new position of every particle, a tree implementation can simplify the problem by agglomerating sets of particles as a single particle, with a mass equal to the sum of masses of the all the particles:. This is the core of the Barnes-Hut algorithm. References on this can be found for example here [http://physics.gmu.edu/%7Elarge/lr_forces/desc/bh/bhdesc.xml], and here [http://www.cita.utoronto.ca/%7Edubinski/treecode/node2.xml]. This method allows us to have a complexity brought down to O(N log N).

In our parallel implementation, we have defined an `Active Object` called `Domain`, which represents a volume in space, and which contains `Planets`. It is either subdivided into smaller `Domains`, or is a leaf of the total tree, and then only contains `Planets`. A Planet is still an Object with mass, velocity and position, but is no longer on a one-to-one connection with a `Domain`. We have cut down communications to the biggest `Domains` possible : when a `Planet` is distant enough, its interactions are not computed, but it is grouped with its local neighbours to a bigger particle. Here is an example of the `Domains` which would be known by the `Domain` drawn in red :



The Domain in the lower left hand-corner, drawn in blue, is also divided into sub-Domains, but this needs not be known by the Domain in red : it assumes all the particles in the blue Domain are only one big one, centered at the center of mass of all the particles within the blue.

In this version, the `Domains` communicate with a reduced set of other `Domains`, spanning on volumes of different sizes. Synchronization is achieved by sending explicitely iteration numbers, and returning when needed older positions. You may notice that some `Domains` seem desynchronized with other ones, having several iterations inbetween. That is no problem because if they then need to be synchronized and send each other information, a mechanism saving the older positions permits to send them when needed.

You may find another view of this example on this web page [http://www-sop.inria.fr/oasis/proactive/apps/nbody-simple.xml].

31.1.9. 9 Conclusion

In this guided tour, we tried to show different facilities provided by ProActive, based on a real problem (nbody). We first saw how to deploy the application, then tuned it by adding Group communication, then removed a bottleneck (due to the hard synchronization) . Finally, given is the code associated to a different algorithm, which cumbersomely shows how to get Active Objects deployed along a tree structure to communicate. Remember that there is another explanation [http://www-sop.inria.fr/oasis/proactive/apps/nbody.xml] of all this on the web.

Chapter 32. 6. Guided Tour Conclusion

This tour was intented to guide you through an overview of ProActive.

You should now be able to start programming with ProActive, and you should also have an idea of the capabilities of the library.

We hope that you liked it and we thank you for your interest in ProActive.

Further information can be found on the website, and suggestion [mailto:proactive-support@inria.fr]s are welcome.

Part VIII. Extending ProActive

Table of Contents

Chapter 33. Adding a Deployment Protocol

33.1. Objectives

ProActive support several deployment protocols. This protocols can be configured through an XML Descriptor file in the **process** section. From time to time, new protocols are added. This documentation describes how to add a new deployment protocol (process) to ProActive.

33.2. Overview

Adding a new process can be divided into two related tasks:

- **Java Process Class**

  In this section, a java class that handles the specific protocol must be implemented. This java class must have certain properties discussed later on.

- **XML Descriptor**

  Since each new protocol requieres different configuration parameteres, the **DescriptorSchema.xsd** and related parsing code must be modified to handle the new process and it's specific parameteres.

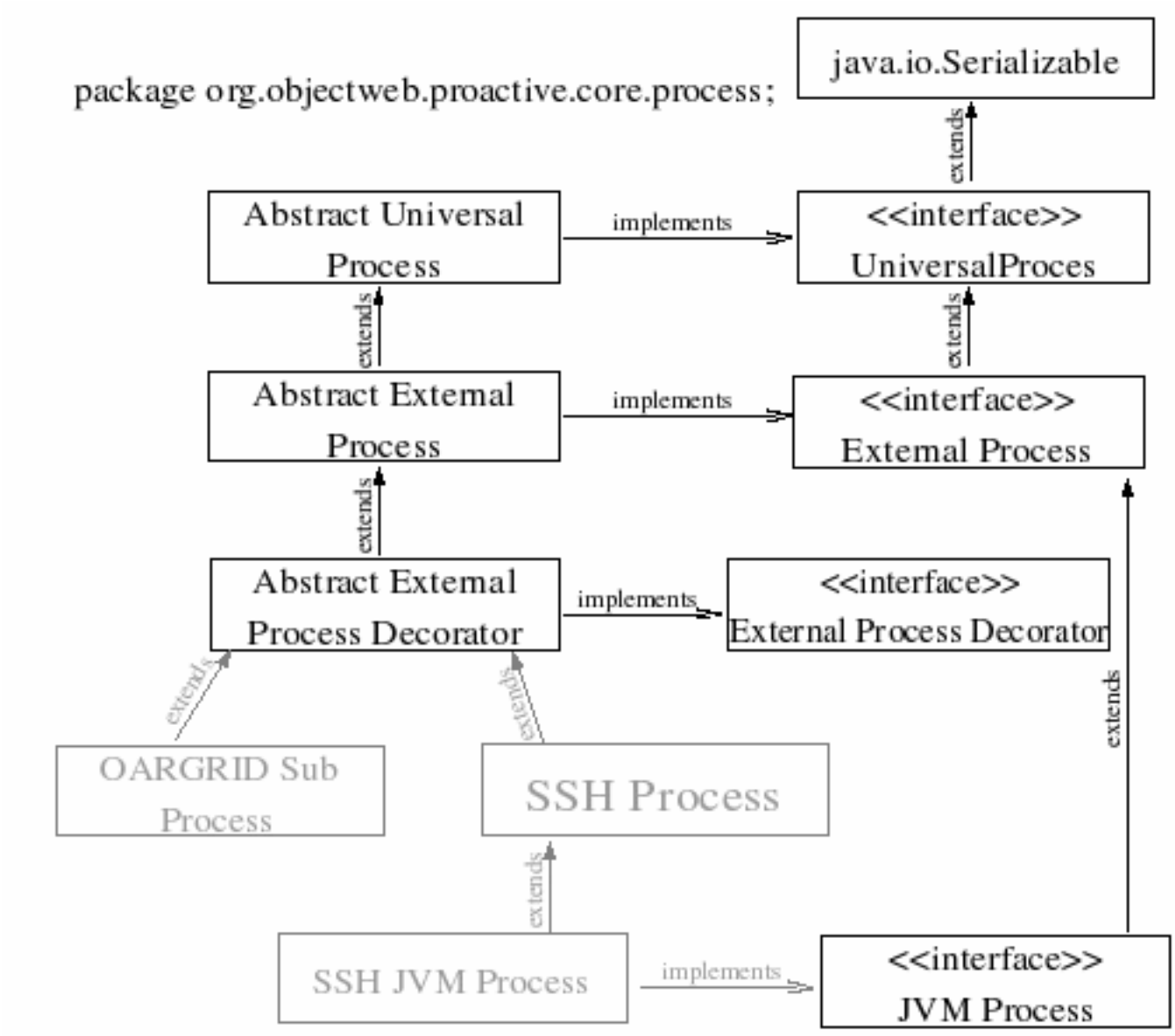
Both of this tasks are closely related because the Java Process Class is used when parsing the Descriptor XML.

33.3. Java Process Class

The Java Process Classes are defined in the **org.objectweb.proactive.core.process** package.

33.3.1. Process Package Arquitecture

Most implementations extend the class **AbstractExternalProcessDecorator**.



In this figure, **OARSubProcess** and **SSHProcess** both extend from **AbstractExternalProcessDecorator**. Notice, that in the case of SSH, more than one class maybe required to succesfully implement the protocol. This is why, every protocol is implemented within it's on directory in the process package:

```
ProActive/src/org/objectweb/proactive/core/process/newprocessdir/
```

Sometimes, implementeing a specific process requiers external libraries, possibly from the original protocol client. The correct place to put this external **.jar** libraries is in:

```
ProActive/lib/newprocessdir/*.jar
```

Before executing a deployment using this new process, don't forget to add this libraries to the **$CLASSPATH** envirorment variable.

33.3.2. The New Process Class

Usualy the new java process class will have a name such as: **ProtocolNameProcess.java**. The **ProtocolNameProcess** class will extend from **AbstractExternalProcessDecorator**. Therefore, at least the following inherited methods must be implemented:

- **public ProtocolNameProcess();**
- **public ProtocolNameProcess(ExternalProcess targetProcess);**
- **public String getProcessId();**
- **public int getNodeNumber();**
- **public UniversalProcess getFinalProcess();**
- **protected String internalBuildCommand();**
- **protected void internalStartProcess(String commandToExecute) throws java.io.IOException;**

33.3.3. The StartRuntime.sh script

On certain clusters, a starting script might be required. Sometimes, this script will be static and receive parameteres at deployment time (globus, pbs, ...), and in other cases it will have to be generated at deployment time (oar, oargrid). In either case, the proper place to put these scipts is:

```
ProActive/scripts/unix/cluster/
```

33.4. XML Descriptor Process

33.4.1. Schema Modifications

The schema file is located at: **ProActive/descriptors/DescriptorSchema.sxd**. This file contains the valid tags allowed in an XML descriptor file.

- **processDefinition Childs**

  The first thing to do, is add the new process tag in:

```
<xs:complexType name="ProcessDefinitionType">
   <xs:choice>
      <xs:element name="jvmProcess" type="JvmProcessType"/>
      <xs:element name="rshProcess" type="RshProcessType"/>
      <xs:element name="maprshProcess" type="MapRshProcessType"/>
      <xs:element name="sshProcess" type="SshProcessType"/>
      <xs:element name="processList" type="ProcessListType"/>
      <xs:element name="processListbyHost" type="ProcessListbyHostType"/>
      <xs:element name="rloginProcess" type="RloginProcessType"/>
      <xs:element name="bsubProcess" type="BsubProcessType"/>
      <xs:element name="pbsProcess" type="PbsProcessType"/>
      <xs:element name="oarProcess" type="oarProcessType"/>
      <xs:element name="oarGridProcess" type="oarGridProcessType"/>
      <xs:element name="globusProcess" type="GlobusProcessType"/>
      <xs:element name="prunProcess" type="prunProcessType"/>
      <xs:element name="gridEngineProcess" type="sgeProcessType"/>
   </xs:choice>
   <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
```

- **Specific Process Tag**

Afterwards, all the tag attributes and subtags need to be defined. In this example, we show the OARGRID tag:

```
<!--oarGridProcess-->
<xs:complexType name="oarGridProcessType">
   <xs:sequence>
      <xs:element ref="processReference"/>
      <xs:element ref="commandPath" minOccurs="0"/>
      <xs:element name="oarGridOption" type="OarGridOptionType"/>
   </xs:sequence>
   <xs:attribute name="class" type="xs:string" use="required"
      fixed="org.objectweb.proactive.core.process.oar.OARGRIDSubProcess"/>

   <xs:attribute name="queue" type="xs:string" use="optional"/>
   <xs:attribute name="bookedNodesAccess" use="optional">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="rsh"/>
            <xs:enumeration value="ssh"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="closeStream" type="CloseStreamType" use="optional"/>

</xs:complexType>
<!--oarGridOption-->
<xs:complexType name="OarGridOptionType">
   <xs:sequence>
      <xs:element name="resources" type="xs:string"/>
      <xs:element name="walltime" type="xs:string" minOccurs="0"/>
      <xs:element name="scriptPath" type="FilePathType" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
```

33.4.2. XML Parsing Handler

33.4.2.1. ProActiveDescriptorConstants.java:

This file is located in **org.objectweb.proactive.core.descriptor.xml** package. It contains the tag names used within XML descriptor files. When adding a new process, new tags should be registered in this file.

33.4.2.2. ProcessDefinitinonHandler.java:

Located in: **org.objectweb.proactive.core.descriptor.xml**, this file is the XML handler for the **process** descriptor section.

- **New XML handler innerclass**

  This class will parse all the process specific tags and attributes. It is an innerclass in the **ProcessDefinitinonHandler.java** file. Sometimes, this class will have a subclass in charge of parsing a subsection of the process tag.

```
protected class OARGRIDProcessHandler extends ProcessHandler {
  public OARGRIDProcessHandler(ProActiveDescriptor proActiveDescriptor) {
    super(proActiveDescriptor);
    this.addHandler(OARGRID_OPTIONS_TAG, new OARGRIDOptionHandler());
  }
  public void startContextElement(String name, Attributes attributes)
    throws org.xml.sax.SAXException {
    super.startContextElement(name, attributes);

    String queueName = (attributes.getValue("queue"));
    if (checkNonEmpty(queueName)) {
      ((OARGRIDSubProcess) targetProcess).setQueueName(queueName);
    }
    String accessProtocol = (attributes.getValue("bookedNodesAccess"));
    if (checkNonEmpty(accessProtocol)) {
      ((OARGRIDSubProcess) targetProcess).setAccessProtocol(accessProtocol)\

;
    }
  }
  protected class OARGRIDOptionHandler extends PassiveCompositeUnmarshaller\

 {
    public OARGRIDOptionHandler() {
      UnmarshallerHandler pathHandler = new PathHandler();
      this.addHandler(OAR_RESOURCE_TAG, new SingleValueUnmarshaller());
      this.addHandler(OARGRID_WALLTIME_TAG, new SingleValueUnmarshaller());
      BasicUnmarshallerDecorator bch = new BasicUnmarshallerDecorator();
      bch.addHandler(ABS_PATH_TAG, pathHandler);
      bch.addHandler(REL_PATH_TAG, pathHandler);
      this.addHandler(SCRIPT_PATH_TAG, bch);
    }
    public void startContextElement(String name, Attributes attributes)
      throws org.xml.sax.SAXException {
    }
    protected void notifyEndActiveHandler(String name, UnmarshallerHandler↩
\
activeHandler)
      throws org.xml.sax.SAXException {
      OARGRIDSubProcess oarGridSubProcess = (OARGRIDSubProcess) targetProce\

ss;
      if (name.equals(OAR_RESOURCE_TAG)) {
        oarGridSubProcess.setResources((String) activeHandler.getResultObje\

ct());
      }
      else if(name.equals(OARGRID_WALLTIME_TAG)){
        oarGridSubProcess.setWallTime((String) activeHandler.getResultObjec\

t());
```

```
      }
      else if (name.equals(SCRIPT_PATH_TAG)) {
        oarGridSubProcess.setScriptLocation((String) activeHandler.getResul\

tObject());
      }
      else {
        super.notifyEndActiveHandler(name, activeHandler);
      }
    }
  }
}
```

- **Registering the new XML handler innerclass**

  The new XML handler innerclass must be registered to handle the parsing of the newprocess tag. This is donde in the constructor:

```
public ProcessDefinitionHandler(ProActiveDescriptor proActiveDescriptor){...}
```

Chapter 34. How to add a new FileTransfer CopyProtocol

FileTransfer protocols can be of two types: **external** or **internal**. Examples of external protocols are: **scp** and **rcp**. While examples of internal protocols are **Unicore** and **Globus**.

Usually external FileTransfer happens before the deployment of the process. On the other hand, internal FileTransfer happens at the same time of the process deployment, because the specific tools provided by the process are used. This implies that internal FileTransfer protocols can not be used with other process (ex: unicore file transfer can not be used when deploying with ssh), but the other way around is valid (ex: scp can be used when deploying with unicore).

34.1. Adding external FileTransfer CopyProtocol

- **Implement the protocol class.** This is done inside the package: **org.objectweb.proactive.core.process.filetransfer;** by extending the abstract class **AbstractCopyProtocol**.
- **FileTransferWorkshop:** Add the name of the protocol to array **ALLOWED_COPY_PROTOCOLS[]**
- **FileTransferWorkshop:** Add the object named based creation to factory method: **copyProtocolFactory(String name){...}**

Note: Choosing the correct name for the protocol is simple, but must be done carefully. All names already in the array ALLOWED_COPY_PROTOCOLS are forbidden. This includes the name **"processDefault"**, which is also forbidden. In some cases **"processDefault"** will correspond to an external FileTransfer protocol (ex: ssh with scp), and in some cases to an internal protocol (ex: unicore with unicore)

34.2. Adding internal FileTransfer CopyProtocol

- Implement the method **protected boolean internalFileTransferDefaultProtocol()** inside the process class. Note that this method will be called if the **processDefault** keyword is specified in the **XML Descriptor Process Section**. Therefore, this method usually must return true, so no other FileTransfer protocols will be tried.
- **FileTransferWorkshop:** Add the name of the protocol to array **ALLOWED_COPY_PROTOCOLS[]**

Note: When adding an internal FileTransfer protocol, **nothing** must be modified or added to the **copyProtocolFactory(){}** method.

Chapter 35. Adding a Fault-Tolerance Protocol

This documentation is a quick overview of how to add a new fault-tolerance protocol within ProActive. A more complete version should be released with the version 3.1. If you wish to get more informations, please feel free to send a mail to proactive@objectweb.org [mailto:proactive@objectweb.org].

35.1. Overview

35.1.1. Active Object side

Fault-tolerance mechanism in ProActive is mainly based on the `org.objectweb.proactive.core.body.ft.protocols` class. This class contains several hooks that are called before and after the main actions of an active object, e.g. sending or receiving a message, serving a request, etc.

For example, with the Pessimistic Message Logging protocol (PML), messages are logged just before the delivery of the message to the active object. Main methods for the FTManager of the PML protocol are then:

```
  /**
   * Message must be synchronously logged before being delivered.
   * The LatestRcvdIndex table is updated
   * @see org.objectweb.proactive.core.body.ft.protocols.FTManager#onDeli\
verReply(org.objectweb.proactive.core.body.reply.Reply)
   */
  public int onDeliverReply(Reply reply) {
     // if the ao is recovering, message are not logged
     if (!this.isRecovering) {
        try {
           // log the message
           this.storage.storeReply(this.ownerID, reply);
           // update latestIndex table
           this.updateLatestRvdIndexTable(reply);
        } catch (RemoteException e) {
           e.printStackTrace();
        }
     }
     return 0;
  }
  /**
   * Message must be synchronously logged before being delivered.
   * The LatestRcvdIndex table is updated
   * @see org.objectweb.proactive.core.body.ft.protocols.FTManager#onRece\
iveRequest(org.objectweb.proactive.core.body.request.Request)
   */
  public int onDeliverRequest(Request request) {
     // if the ao is recovering, message are not logged
     if (!this.isRecovering) {
        try {
           // log the message
           this.storage.storeRequest(this.ownerID, request);
           // update latestIndex table
           this.updateLatestRvdIndexTable(request);
        } catch (RemoteException e) {
           e.printStackTrace();
```

```
         }
      }
      return 0;
   }
```

The local variable `this.storage` is remote reference to the checkpoint server. The FTManager class contains a reference to each fault-tolerance server: fault-detector, checkpoint storage and localization server. Those reference are initialized during the creation of the active object.

A FTManager must define also a `beforeRestartAfterRecovery()` method, which is called when an active object is recovered. This method usually restore the state of the active object so as to be consistent with the others active objects of the application.

For example, with the PML protocol, all the messages logged before the failure must be delivered to the active object. The method `beforeRestartAfterRecovery()` thus looks like:

```
   /**
    * Message logs are contained in the checkpoint info structure.
    */
   public int beforeRestartAfterRecovery(CheckpointInfo ci, int inc) {
      // recovery mode: received message no longer logged
      this.isRecovering = true;
       //get messages
      List replies = ci.getReplyLog();
      List request = ci.getRequestLog();
       // add messages in the body context
      Iterator itRequest = request.iterator();
      BlockingRequestQueue queue = owner.getRequestQueue();
      // requests
      while (itRequest.hasNext()) {
         queue.add((Request) (itRequest.next()));
      }
      // replies
      Iterator itReplies = replies.iterator();
      FuturePool fp = owner.getFuturePool();
      try {
         while (itReplies.hasNext()) {
            Reply current = (Reply) (itReplies.next());
            fp.receiveFutureValue(current.getSequenceNumber(),
               current.getSourceBodyID(), current.getResult(), current\
);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
      // normal mode
      this.isRecovering = false;
      // enable communication
      this.owner.acceptCommunication();
      try {
         // update servers
         this.location.updateLocation(ownerID, owner.getRemoteAdapter())\
;
         this.recovery.updateState(ownerID, RecoveryProcess.RUNNING);
      } catch (RemoteException e) {
         logger.error("Unable to connect with location server");
```

```
            e.printStackTrace();
        }
        return 0;
    }
```

The parameter `ci` is a `org.objectweb.proactive.core.body.ft.checkpointing.CheckpointInfo`. This object contains all the informations linked to the checkpoint used for recovering the active object, and is used to restore its state. The programmer might defines his own class implementing `CheckpointInfo`, to add needed informations, depending on the protocol.

35.1.2. Server side

ProActive include a global server that provide fault detection, active object localization, resource service and checkpoint storage. For developing a new fault-tolerance protocol, the programmer might specify the behavior of the checkpoint storage by extending the class `org.objectweb.proactive.core.body.ft.servers.storage.CheckpointServerImpl`. For example, only for the PML protocol and not for the CIC protocol, the checkpoint server must be able to log synchronously messages. The other parts of the server can be used directly.

To specify the recovery algorithm, the programmer must extends the `org.objectweb.proactive.core.body.ft.server` In the case of the CIC protocol, all the active object of the application must recover after one failure, while only the faulty process must restart with the PML protocol; this specific behavior is coded in the recovery process.

Chapter 36. MOP : Metaobject Protocol

36.1. Implementation: a Meta-Object Protocol

ProActive is built on top of a metaobject protocol (MOP) that permits reification of method invocation and constructor call. As this MOP is not limited to the implementation of our transparent remote objects library, it also provides an open framework for implementing powerful libraries for the Java language.

As for any other element of ProActive, this MOP is entirely written in Java and does not require any modification or extension to the Java Virtual Machine, as opposed to other metaobject protocols for Java {Kleinoeder96}. It makes extensive use of the Java Reflection API, thus requiring JDK 1.1 or higher. JDK 1.2 is required in order to suppress default Java language access control checks when executing reified non-public method or constructor calls.

36.2. Principles

If the programmer wants to implement a new metabehavior using our metaobject protocol, he or she has to write both a concrete (as opposed to abstract) class and an interface. The concrete class provides an implementation for the metabehavior he or she wants to achieve while the interface contains its declarative part.

The concrete class implements interface **Proxy** and provides an implementation for the given behavior through the method **reify**:

```
public Object reify (MethodCall c) throws Throwable;
```

This method takes a reified call as a parameter and returns the value returned by the execution of this reified call. Automatic wrapping and unwrapping of primitive types is provided. If the execution of the call completes abruptly by throwing an exception, it is propagated to the calling method, just as if the call had not been reified.

The interface that holds the declarative part of the metabehavior has to be a subinterface of `Reflect` (the root interface for all metabehaviors implemented using ProActive). The purpose of this interface is to declare the name of the proxy class that implements the given behavior. Then, any instance of a class implementing this interface will be automatically created with a proxy that implements this behavior, provided that this instance is not created using the standard `new` keyword but through a special static method: `MOP.newInstance`. This is the only required modification to the application code. Another static method, `MOP.newWrapper`, adds a proxy to an already-existing object; the `turnActive` function of ProActive, for example, is implemented through this feature.

36.3. Example of a different metabehavior: EchoProxy

Here's the implementation of a very simple yet useful metabehavior : for each reified call, the name of the invoked method is printed out on the standard output stream and the call is then executed. This may be a starting point for building debugging or profiling environments.

```
class EchoProxy extends Object implements Proxy {
  // here are constructor and variables declaration
  // [...]
  public Object reify (MethodCall c) throws Throwable {
```

```
        System.out.println (c.getMethodName());
        return c.execute (targetObject);
      }
    }
    interface Echo extends Reflect {
      public String PROXY_CLASS= "EchoProxy";
    }
```

36.3.1. Instantiating with the metabehavior

Instantiating an object of any class with this metabehavior can be done in three different ways: instantiation-based, class-based or object-based. Let's say we want to instantiate a `Vector` object with an `Echo` behavior.

- Standard Java code would be :

```
Vector v = new Vector(3);
```

- ProActive code, with instantiation-based declaration of the metabehavior (the last parameter is `null` because we do not have any additional parameter to pass to the proxy) :

```
Object[] params = {new Integer (3)};
Vector v = (Vector) MOP.newInstance("Vector", params, "EchoProxy", null);
```

- with class-based declaration :

```
public class MyVector extends Vector implements Echo {}
Object[] params = {new Integer (3)} ;
Vector v = (Vector) MOP.newInstance("Vector", params, null);
```

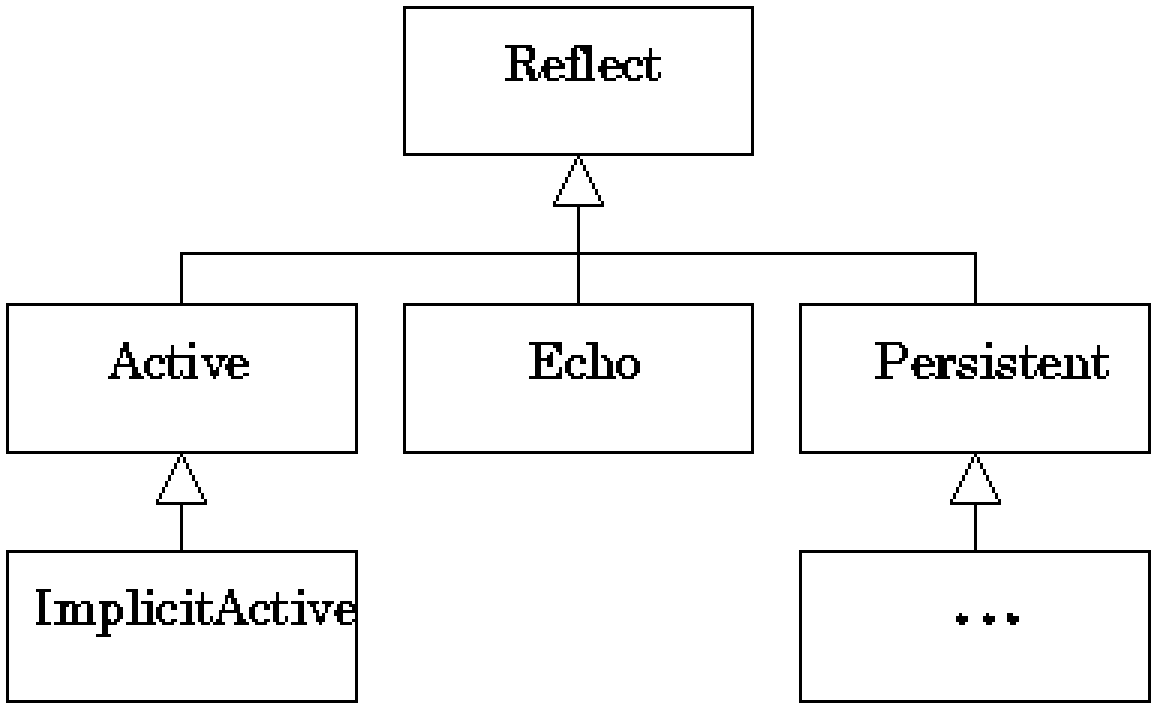
- with object-based declaration :

```
Vector v = new Vector (3);
v=(Vector) MOP.newWrapper("EchoProxy",v);
```

This is the only way to give a metabehavior to an object that is created in a place where we cannot edit source code. A typical example could be an object returned by a method that is part of an API distributed as a JAR file, without source code. Please note that, when using `newWrapper`, the invocation of the constructor of the class `Vector` is not reified.

36.4. The Reflect interface

All the interfaces used for declaring *metabehaviors* inherit directly or indirectly from `Reflect`. This leads to a hierarchy of metabehaviors such as shown in the figure below.



Reflect Interface and sub-interfaces diagram

Note that `ImplicitActive` inherits from `Active` to highlight the fact that implicit synchronization somewhere always relies on some hidden explicit mechanism. Interfaces inheriting from `Reflect` can thus be logically grouped and assembled using multiple inheritance in order to build new metabehaviors out of existing ones.

36.5. Limitations

Due to its commitment to be a 100% Java library, the MOP has a few limitations:

- Calls sent to instances of final classes (which includes all arrays) cannot be reified.
- Primitive types cannot be reified because they are not instance of a standard class.
- Final classes (which includes all arrays) cannot be reified because they cannot be subclassed.