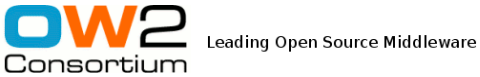
An Open Source Solution for Enterprise Grids & Clouds

ProActive Programming

Reference Manual

Version 2014-02-17

ActiveEon Company, in collaboration with INRIA



Generated on 2014-02-17

ProActive Programming v2014-02-17 Documentation

Legal Notice

Contact: proactive@ow2.org or contact@activeeon.com

Mailing List

proactive@ow2.org

Mailing List Archive

http://www.objectweb.org/wws/arc/proactive

Bug-Tracking System

http://bugs.activeeon.com/browse/PROACTIVE

Contributors and Contact Information

Team Leader

Denis Caromel
INRIA 2004, Route des Lucioles, BP 93
06902 Sophia Antipolis Cedex
France
phone: +33 492 387 631
fax: +33 492 387 971
e-mail: Denis.Caromel@inria.fr

Contributors from OASIS Team	Contributors from ActiveEon Company
Brian Amedro	Vladimir Bodnartchouk
Francoise Baude	Arnaud Contes
Francesco Bongiovanni	Cédric Dalmasso
Florin-Alexandru Bratu	Christian Delbé
Viet Dung Doan	Arnaud Gastinel
Yu Feng	Jean-Michel Guillaume
Imen Filali	Olivier Helin
Fabrice Fontenoy	Clément Mathieu
Ludovic Henrio	Maxime Menant
Fabrice Huet	Emil Salageanu
Elaine Isnard	Jean-Luc Scheefer
Vasile Jureschi	Mathieu Schnoor
Muhammad Khan	
Virginie Legrand Contes	
Eric Madelaine	
Elton Mathias	
Paul Naoumenko	
Laurent Pellegrino	
Guilherme Peretti-Pezzi	
Franca Perrina	
Marcela Rivera	
Christian Ruz	
Bastien Sauvan	
Oleg Smirnov	
Marc Valdener	
Fabien Viale	

Former Important Contributors

Laurent Baduel (Group Communications)
Vincent Cave (Legacy Wrapping)
Alexandre di Costanzo (P2P, B&B)
Abhijeet Gaikwad (Option Pricing)
Mario Leyton (Skeleton)
Matthieu Morel (Initial Component Work)
Romain Quilici
Germain Sigety (Scheduling)
Julien Vayssiere (MOP, Active Objects)

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

In order to make the ProActive Programming documentation easier to read, it has been split into four manuals:

- **ProActive Get Started Manual** - This manual contains an overview of ProActive Programming showing different examples and explaining how to install the middleware. It also includes a tutorial teaching how to use it. This manual should be the first one to be read for beginners.
- **ProActive Reference Manual** - This manual is the main manual where the concepts of ProActive are described. Information on configuration and deployment are described in that manual. It also includes guides for high-level APIs usage such as Master-Worker, SMPD APIs.
- **ProActive Advanced Features Manual** - This manual describes some advanced features like Fault-Tolerance, ProActive Compile-Time Annotations or Web Services Exportation. In Addition, it gives information on some other high-level APIs such as Monte-Carlo or Branch and Bound APIs. Finally, it helps advanced user to extend ProActive.
- **ProActive Components Manual** - ProActive defines a component model called ProActive/GCM suitable to support the development of efficient grid applications. This manual therefore contains all necessary information to be able to understand and use this component model. This model is really linked to ProActive so a ProActive/GCM user may have to refer to the ProActive manuals form time to time.

These manuals should be read in the order defined above. However, this is not essential as these documentations are linked together. Besides, if some links seems to be dead in one of these manuals, make sure that all of them had been built previously (multiple html version). So as to build all the manuals at once, go to your ProActive compile/ directory and type build[.bat] doc.ProActive.manualHtml. This builds all manuals in all formats. You may also need the javadoc documentation since these manuals sometime refer to it. To build all the javadoc documentations, type build[.bat] doc.ProActive.javadoc.complete doc.ProActive.javadoc.published inside the compile/ directory. If you just want to build one of these manuals in a specific format, type build[.bat] to see all the possible targets and chose the one you are interested in.

Part I. Programming With Active Objects

Table of Contents

Chapter 1. Active Object Definition

1.1. Overview

In this part, we will present the process of creating an active object from a class or from another already existing object. We will also look at the internal structure of an active object and at its thread. In the last chapters, we will examine group communication, mobile agents and exception handling for active objects.

1.2. Active Object Structure

Active objects are the basic units of activity and distribution used for building concurrent applications using ProActive. As opposed to passive (regular) objects, an active object has its own thread and execution queue. ProActive manages active object threads relieving programmers from explicitly manipulating Thread objects and thus, making the use of threads transparent.

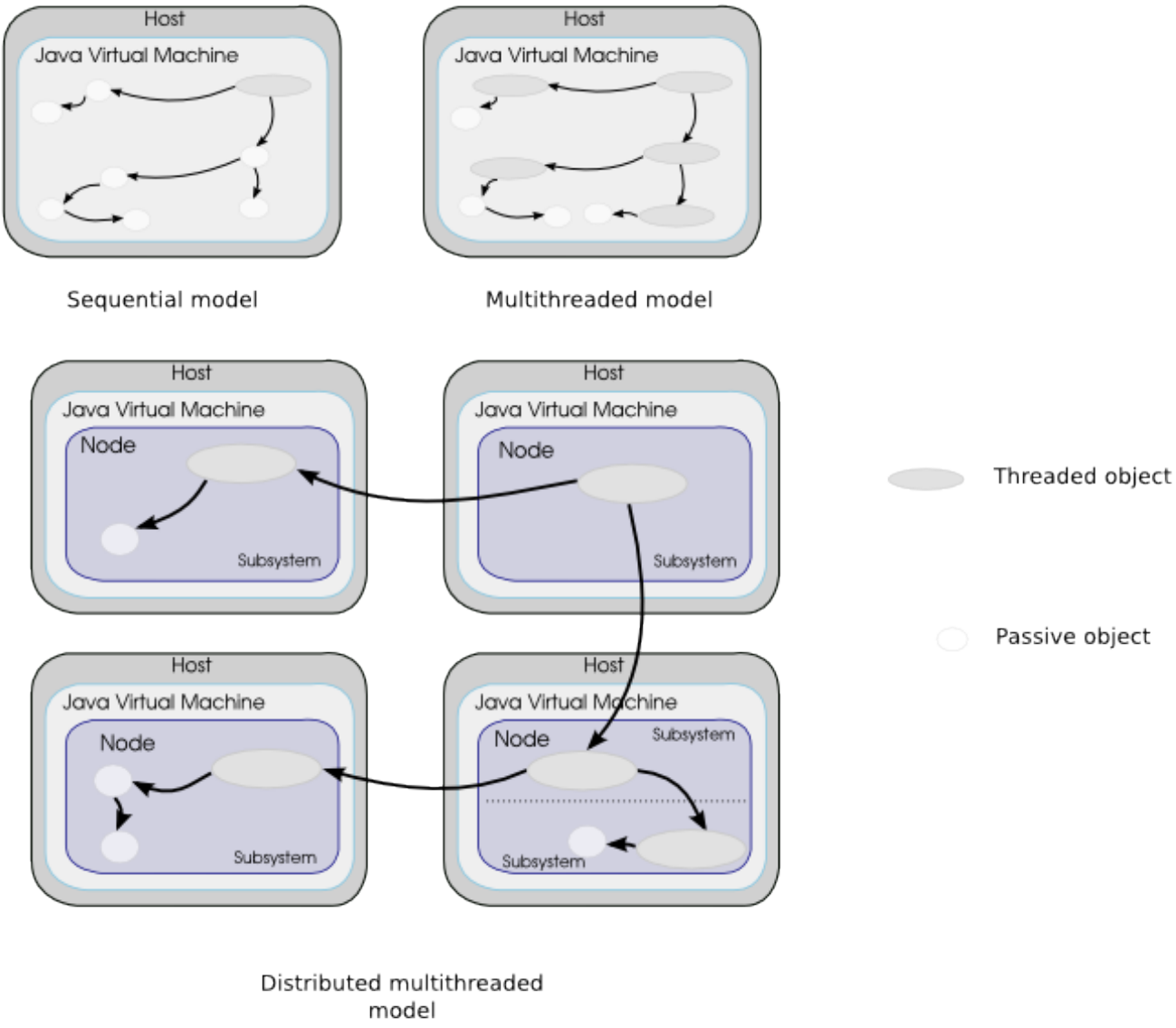


Figure 1.1. Three different computational models

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 runs with its own thread) and its location (local or remote) are transparent. As a matter of fact, any active object can be manipulated just like if it were a passive instance of the same class.

An application based on active objects is structured in subsystems. A subsystem is composed of only one active object (with its own thread) and several passive objects (possibly zero). The active object thread executes only the methods invoked on the active object by other active objects or by passive objects of the subsystem of the active object.

This has 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 is also applied to the 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 from a passive object. Then, the calling thread can continue executing its code just as 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 type of inter-object synchronization policy is known as **wait-by-necessity**.

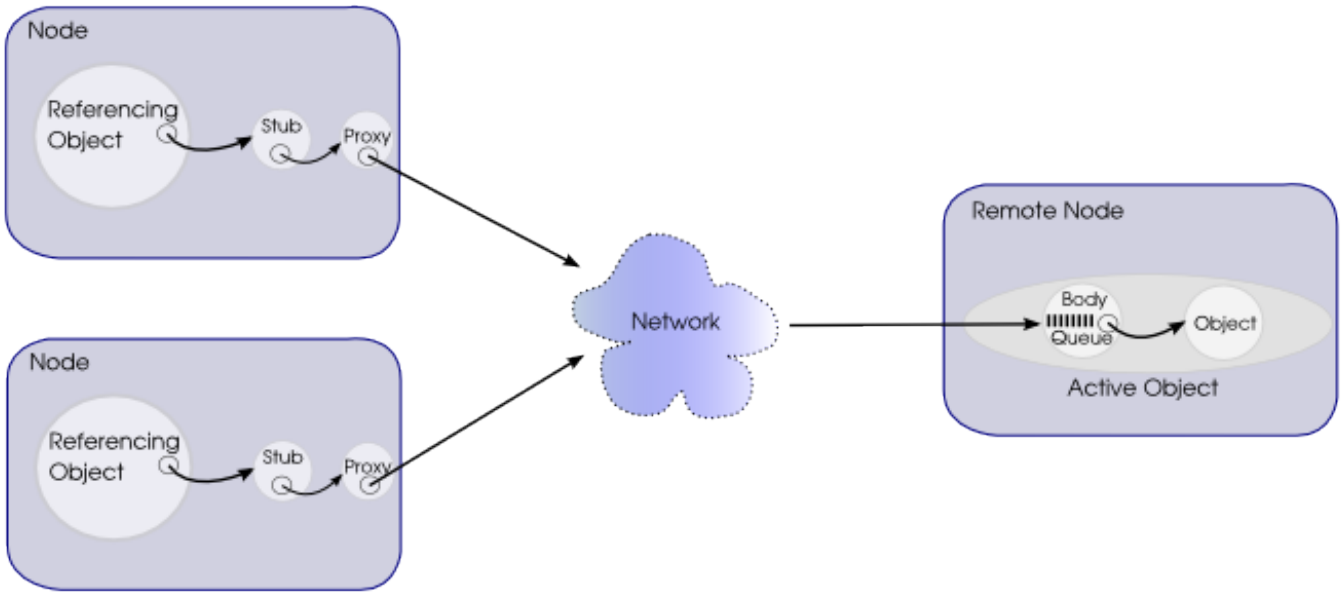


Figure 1.2. Proactive Active Object structure

An active object is the composition of two objects:

- a body (Body)
- a regular instance of the Object

The **body** is responsible for receiving calls on the active object, storing these calls in the **queue** of pending calls (also called requests). It will execute these calls in an order specified by a specific synchronization policy. If no specific synchronization policy is provided, calls are managed in a FIFO order (First In, First Out, in other word, the first arrived request is the first one to be served). The body is not visible from the outside of its active object. Thus, an active object looks exactly like a standard object from the user's perspective. 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 and post conditions and class invariants.

On the side of the subsystem sending a call to an active object, the active object is accessed through a **stub** and a **proxy**.

The main responsibilities of the **proxy** are to generate future objects for representing future values and perform deep-copy of passive objects passed as parameters. Passive objects are not shared between subsystems. Any call on an remote active object using passive objects as arguments leads to a deep-copy of the passive objects for the subsystem of the remote active object.

As for the role of the **stub**, it is in charge of reifying all the method calls that can be performed through a reference to the active object. Reifying a call simply means constructing an object (in our case, all reified calls are instance of class MethodCall) that represents the call, so that it can be manipulated as any other object. This reified call is then processed by the other active object components in order to achieve the behavior we expect from an active object.

However, the use of the **stub**, **proxy**, **body**, and **queue** is transparent. ProActive manages all of them. Users access to active objects in the same way as passive objects.

Chapter 2. Active Objects: Creation And Advanced Concepts

2.1. Overview

Active objects are created on a per-object basis: an application can contain active as well as passive instances of a given class. In this section, we will present the three methods for creating active instances of classes and how to use arguments passed to the methods to control creation. Although almost any object can be turned into an Active Object, there are some restrictions that will be detailed below. We will also present a deep explanation of the active object structure and behaviour. Examples in this chapter contain only the code necessary for the creation of active objects. To see how to start nodes, JVMs, and virtual nodes on which objects can be instantiated, please refer to Chapter 11, *ProActive Basic Configuration*

In **ProActive**, there are two ways to create active objects: instantiation based creation and by using an existing object. Instantiation based creation is done through the PAActiveObject.newActive(...) and the PAActiveObject.newActiveInParallel(...) method whereas object based creation is done through the PAActiveObject.turnActive(...) method. These methods are part of the org.objectweb.proactive.api package. Each method takes several parameters that indicate how the active object is created and most of the parameters are common among creation methods. All the available creation methods with parameters are presented in the paactiveobject javadoc [1].

For the examples described in this chapter, we will use the following class that can be instantiated as an active object:

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
import org.objectweb.proactive.Body;
import org.objectweb.proactive.core.util.wrapper.IntWrapper;


/**
 * @author ProActive Team
 *
 * Class used for documentation example, in particular
 * for Chapter 9. Active Objects: Creation And Advanced Concepts.
 */
public class Worker implements Serializable, InitActive {

   private IntWrapper age = new IntWrapper(0);
   private String name = "Anonymous";

   /**
    * Empty no-arg constructor needed by ProActiv
    */
   public Worker() {
   }

   /**
    * Constructor with arguments
    *
    * @param age
```

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../api_published/org/objectweb/proactive/api/PAActiveObject.html

```
     * @param name
     */
    public Worker(IntWrapper age, String name) {
        this.setAge(age);
        this.setName(name);
    }

}
```

2.2. Restrictions on creating active objects

Not all classes can be used to instantiate active objects. There are some restrictions, most of them being caused by the 100% Java compliance, which forbids the modification of the Java Virtual Machine and of the compiler.

Some of these restrictions are at class-level such as:

- final classes cannot be used to instantiate active object
- non public classes cannot be used to instantiate active object
- classes without a no-argument constructor cannot be reified

Member classes (i.e. non-static named internal classes) can be used to instantiate active objects only through the PAActiveObject.turnActive(...) method (see Section 2.4, “Object Based Creation using PAActiveObject.turnActive(...)”). In other words, an instance of a member class can only be activated after creation, and not directly created as an active object. Attempting to directly create an active object from a member class with PAActiveObject.newActive(...) method will throw a NotActivatableException. This is mainly due to the fact that active objects created from the same member class must share a reference to the enclosing instance: as PAActiveObject.newActive(...) performs a deep copy of the activated instance, the enclosing instance would be a different copy for each active objects. Because of this sharing, active objects created from a member class cannot be migrated (Chapter 4, *Mobile Agents And Migration*) nor turned as fault-tolerant active object (Chapter 32. Fault-Tolerance[2]). Attempting to migrate or checkpoint such an active object will throw a ProActiveException.

There are also restrictions at method level within a class. Final methods cannot be used because the stub is created from the object and having methods final prevents the stub from overriding the methods.

2.3. Instantiation Based Creation

For creating new instances of active objects, we can either use the PAActiveObject.newActive(...) method or PAActiveObject.newActiveInParallel(...) one. ProActive also provides a way to create multiple active objects in parallel on several nodes. PAActiveObject.newActiveInParallel(...) creates a number of active objects deployed on one or more nodes. The object creation is optimized by a thread pool.

When using instantiation based creation, any argument passed to the constructor of the reified object through PAActiveObject.newActive(...) or PAActiveObject.newActiveInParallel(...) is serialized and passed by copy to the object. That is because the model behind **ProActive** is uniform whether the active object is instantiated locally or remotely. Parameters are therefore guaranteed to be passed by copy to the constructor. When using PAActiveObject.newActive(...), one needs to make sure that the constructor arguments are Serializable. On the other hand, the class used to create the active object does not need to be Serializable even in the case of remotely-created Active Objects. Bear in mind also that a reified object must have a declared empty no-args constructor in order to be properly created.

2.3.1. Using PAActiveObject.newActive(...)

To create a single active object from the class Worker in the local JVM, we use the code hereafter. If the invocation of the constructor of class Worker throws an exception, it is placed inside an exception of type ActiveObjectCreationException. When the call to newActive returns, the active object has been created and its active thread is started.

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/faultTolerance.html

```
// Set the constructor parameters
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };

Worker charlie;
try {
    charlie = PAActiveObject.newActive(Worker.class, params);
} catch (ActiveObjectCreationException aoExcep) {
    // the creation of ActiveObject failed
    System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
    System.err.println(nodeExcep.getMessage());
}
```

2.3.2. Using PAActiveObject.newActiveInParallel(...)

The following code deploys an active object on nodes contained in the virtual node VN (current nodes). In this case, the Worker constructor does not take any arguments. However, we have to create an Object array whose length is equal to the number of nodes. The creation of active objects is optimized by a thread pool. When the call to newActiveInParallel returns, the active objects have been created and their threads have been started.

```
/***** GCM Deployment *****/
File applicationDescriptor = new File(gcmaPath);

GCMApplication gcmad;
try {
    gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
} catch (ProActiveException e) {
    e.printStackTrace();
    return;
}
gcmad.startDeployment();

// Take 2 nodes from the available nodes of VN
GCMVirtualNode vn = gcmad.getVirtualNode("VN");
vn.waitReady();
Node[] nodes = vn.getCurrentNodes().toArray(new Node[(int) vn.getNbCurrentNodes()]);
/*************************/

try {
    Worker[] workers = (Worker[]) PAActiveObject.newActiveInParallel(Worker.class.getName(),
            new Object[nodes.length][], nodes);
} catch (ClassNotFoundException classExcep) {
    System.err.println(classExcep.getMessage());
}
```

2.4. Object Based Creation using PAActiveObject.turnActive(...)

Object based creation is used for turning an existing passive object instance into an active one. It has been introduced in ProActive as an answer to the problem of creating active objects from already existing objects for which we do not have access to the source code.

Since the object already exists before turning it active, there is no serialization involved when we create the object. When we invoke PAActiveObject.turnActive on the object, two cases are possible. If we create the active object locally (on a local node), it will not be serialized. If we create the active object remotely (on a remote node), the reified object will be serialized. Thus, if the turnActive is done on a remote node, the class used to create the active object this way has to be Serializable. In addition, when using turnActive ,

care must be taken that no other references to the originating object are kept by other objects after the call to turnActive. A direct call to a method of the originating object without passing by a ProActive stub on this object will break the ProActive model.

The simplest code for object based creation looks like this:

```
Worker charlie = new Worker(new IntWrapper(19), "Charlie");
try {
   charlie = (Worker) PAActiveObject.turnActive(charlie);
} catch (ActiveObjectCreationException aoExcep) {
   // the creation of ActiveObject failed
   System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
   System.err.println(nodeExcep.getMessage());
}
```

In this case, the active object is created locally in the current node. However, you can specify a second parameter which is the location where the active object will be created.

When using this method, the programmer has to make sure that no other reference on the passive object exist after the call to PAActiveObject.turnActive(...). If such references are used for calling methods directly on the passive object (without going through its stub, proxy, and body), the model will not be consistent and specialization of synchronization will no be guaranteed.

2.5. Active Object Creation Arguments

2.5.1. Using classname and target

PAActiveObject.newActive(...) and PAActiveObject.newActiveInParallel(...) always take as a first argument the class name from which the active object will be instantiated. The classname argument must be of type java.lang.String and its value is usually obtained by calling ClassToInstantiateFrom.class.getName(). PAActiveObject.turnActive(...) does not create an active object from a class but from an existing object, therefore it takes as a first argument the object to be turned active.

2.5.2. Using Constructor Arguments

In order to create the active object, PAActiveObject.newActive(...) and PAActiveObject.newActiveInParallel(...) have to take a list of constructor arguments to be passed to the constructor of the class to be instantiated as an active object. Arguments to the constructor of the class have to be passed as an array of Object. Also, we have to make sure that the constructor arguments are Serializable since they are passed by a serialized copy to the object. The ProActive runtime determines which constructor of the class to call according to the type of the elements of this array. Nevertheless, there is still room for some ambiguity in resolving the constructor because as the arguments of the constructor are stored in an array of type Object[] or Object[][]. If one argument is null the runtime can obviously not determine its type. In this case a exception is thrown specifying that ProActive cannot determine the constructor. In the example below, an ambiguity exists between the two constructors if the corresponding element of the Object array is null

```
// Passing null as
// an argument for the constructor
// leads to the generation of an exception
public Worker(IntWrapper age) {
   this.setAge(age);
}

public Worker(String name) {
   this.setName(name);
}
```

If we use PAActiveObject.newActiveInParallel(..., Nodes[] nodes) we have the make sure that the length of the first dimension of java.lang.Object[][] is equal to the number of nodes since an active object will be deployed on each node. The second dimension contains the number of the constructor arguments for each deployed active object. Different active objects can have different constructor arguments.

2.5.3. Using Parameterized Classes

If you want to instantiate a generic object as an active object, you have to specify its type parameters into the constructor methods. For instance, let's consider the generic class Pair hereafter:

```
package org.objectweb.proactive.examples.documentation.classes;

public class Pair<X, Y> {
   private X first;
   private Y second;

   public Pair() {
   }

   public Pair(X a1, Y a2) {
      first = a1;
      second = a2;
   }

   public X getFirst() {
      return first;
   }

   public Y getSecond() {
      return second;
   }

   public void setFirst(X arg) {
      first = arg;
   }

   public void setSecond(Y arg) {
      second = arg;
   }
}
```

Pair is a generic class which requires two type parameters. When we create an active object using this class, we have to specify these two types. This is done through the second argument of the PAActiveObject.newActive() method which has to be an array of classes. In that case, the array of constructor arguments is therefore moved to the third position of the arguments of PAActiveObject.newActive().

Thus, instantiation of a Pair active object will look like this:

```
Worker charlie = new Worker(new IntWrapper(30), "Charlie");
try {
   // Declaration of the array of Class representing
   // type parameters
   Class<?>[] typeParameters = new Class<?>[] { Worker.class, String.class };

   // Declaration of the constructor parameters
```

```
    Object[] constructorParameters = new Object[] { charlie, "Researcher" };

    // Instantiation of the active object
    Pair<Worker, String> pair = (Pair<Worker, String>) PAActiveObject.newActive(Pair.class.getName(),
            typeParameters, constructorParameters);

} catch (ActiveObjectCreationException e) {
    e.printStackTrace();
} catch (NodeException e) {
    e.printStackTrace();
}
```

As you can see, we have created an active object of Pair whose type parameters are Worker and String. As the constructor used for this creation takes the two type parameters as parameters, we gave a Worker (first type parameter) and a String to the PAActiveObject.newActive() method.

We have presented this particularity using the PAActiveObject.newActive() method but obviously, it can also be applied to the PAActiveObject.newActiveInParallel() method as well as to the PAActiveObject.turnActive() method.

2.5.4. Using A Node

In order to create the new active object on a specific (remote) JVM, we can also give a node as parameter to newActive method. The JVM is identified using a Node object or a String that contains an URL pointing to the node. If this parameter is not given, the active object is created in the current JVM and is attached to a default Node. The node argument or the URL string can be used with PAActiveObject.newActive(...) and PAActiveObject.turnActive(...). As for the PAActiveObject.newActiveInParallel(..) method, we can specify nodes using an array of Node since we create several active objects. In that case, an active object will be created on each node contained in this array.

```
/***** GCM Deployment *****/
File applicationDescriptor = new File(gcmaPath);

GCMApplication gcmad;
try {
    gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
} catch (ProActiveException e) {
    e.printStackTrace();
    return;
}
gcmad.startDeployment();

// Take a node from the available nodes of VN
GCMVirtualNode vn = gcmad.getVirtualNode("VN");
vn.waitReady();
Node node = vn.getANode();
/*************************/

// Set the constructor parameters
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };

Worker charlie;
try {
    charlie = PAActiveObject.newActive(Worker.class, params, node);
} catch (ActiveObjectCreationException aoExcep) {
    // the creation of ActiveObject failed
```

```
    System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
    System.err.println(nodeExcep.getMessage());
}
```

To deploy using the URL of a node, we just have to replace the node previously used by its URL:

```
String nodeURL = node.getNodeInformation().getURL();
charlie = PAActiveObject.newActive(Worker.class, params, nodeURL);
```

If you want to use the PAActiveObject.turnActive(...) method instead of the PAActiveObject.newActive(...) one, you can proceed exactly in the same way as previously.

In order to deploy several active objects with PAActiveObject.newActiveInParallel(...) , we can only pass an array of Node (see the example in Section 2.3.2, " Using PAActiveObject.newActiveInParallel(...) ").

2.5.5. Using A Custom Activity

Customizing the activity of the active object is at the core of ProActive because it allows to specify fully the behavior of an active object. By default, an object turned into an active object serves its incoming requests in a FIFO manner. In order to specify another policy for serving the requests or to specify any other behaviors, we can implement interfaces defining methods that will be automatically called by ProActive.

It is possible to specify what to do before the activity starts, what the activity is and what to do after it ends. The three steps are:

- the initialization of the activity (done only once)
- the activity itself
- the end of the activity (done only once)

Three interfaces are used to define and implement each step:

- InitActive - contains the initActivity(Body body) method to be implemented
- RunActive - contains the runActivity(Body body) method to be implemented
- EndActive - contains the endActivity(Body body) method to be implemented

In case of a migration, an active object stops and restarts its activity automatically without invoking the initialization or ending phases. Only the activity itself is restarted.

They are two ways to define each of the three phases of an active object:

- implementing one or more of the three interfaces directly in the class used to create the active object
- passing an object implementing one or more of the three interfaces in parameter to the method newActive, newActiveInParallel or turnActive (parameter activity of the type Active in those methods).

Note that methods defined by those 3 interfaces are guaranteed to be called by the active thread of the active object.

2.5.5.1. Algorithms deciding which activity to invoke

Activity invocations follow the algorithm hereafter (activity is the object passed as a parameter to newActive or turnActive):

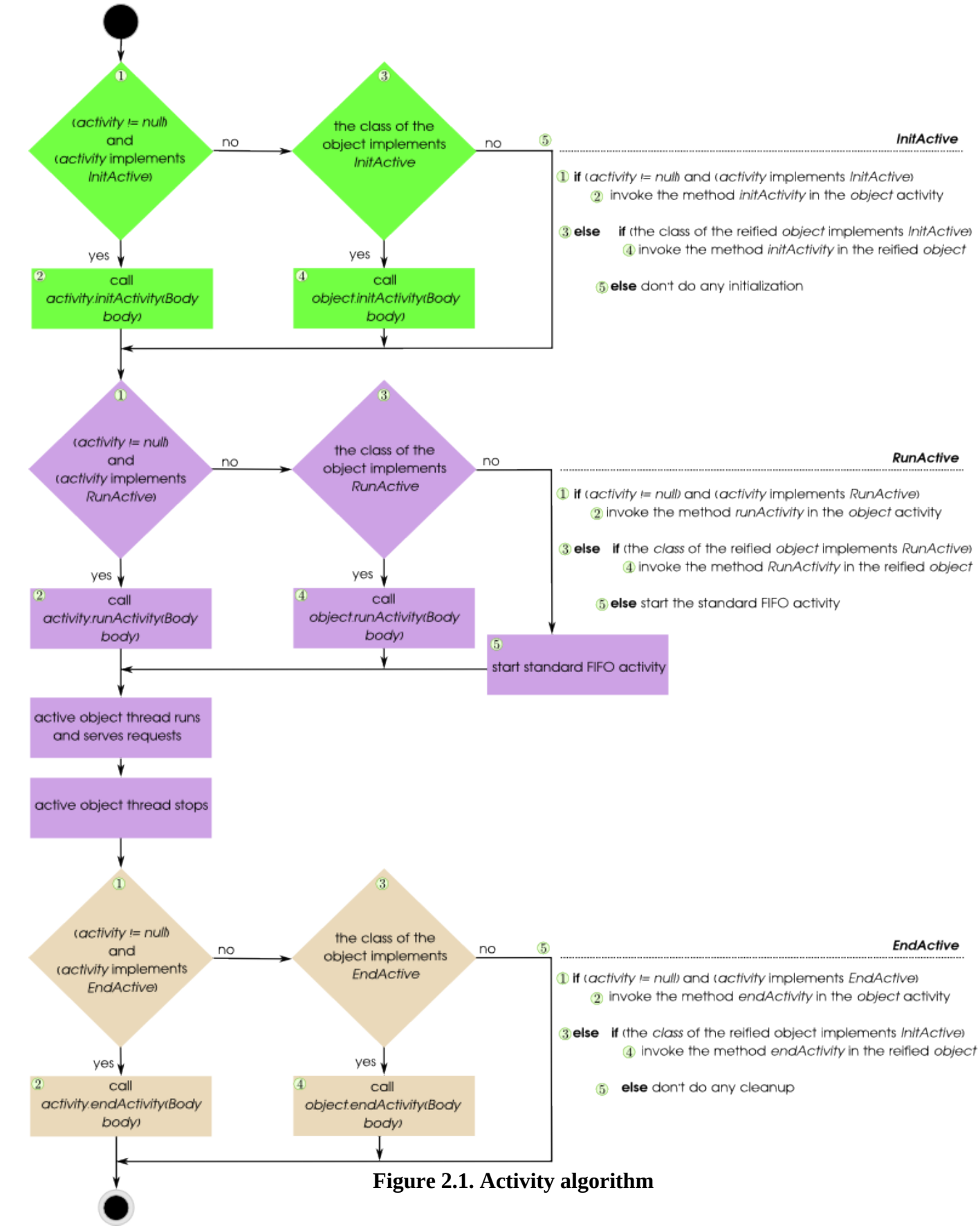


Figure 2.1. Activity algorithm

When the active object is about to be instantiated, if an activity object has been given to the creation method and if it implements the initActivity method, then this method is executed. Else, if the object we want to activate implements this method, then this one is executed. Otherwise, no method is executed before the active object creation.

In the same way, the activity itself used for the object is first the one defined in the activity object (implementing the runActivity method). Then, if no runActivity method has been defined in an activity object, the one defined in the object itself is executed. Otherwise, the standard activity phase will be used (which serves requests in a FIFO order).

Finally, when terminating the active object, the same scenario occurs. In that case, as it was for the initialising phase, if no endActivity method has been defined anywhere, no method will be executed. In other word, there is no default behaviour for ending.

Note

If an activity method (initActivity, runActivity or endActivity) is implemented both in an activity object passed as parameter to the creation method and also in the object we want to activate, then it is the method defined in the activity object that will be executed. This behaviour enables in particular to override activity phases of an object that we do not have access to its code (or that we do not want to modify).

2.5.5.2. Implementing the interfaces directly in the class

Implementing the interfaces directly in the class used to create the active object is the easiest solution when you control the class that you make active. Depending on which phase in the life of the active object you want to customize, you have to implement the corresponding interface (one or more): InitActive, RunActive and EndActive. Here is the Worker into which we have customized its initialization and its ending.

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
import org.objectweb.proactive.Body;
import org.objectweb.proactive.core.util.wrapper.IntWrapper;

import org.objectweb.proactive.InitActive;



/**
 * @author ProActive Team
 *
 * Class used for documentation example, in particular
 * for Chapter 9. Active Objects: Creation And Advanced Concepts.
 */
public class Worker implements Serializable, InitActive {

   private IntWrapper age = new IntWrapper(0);
   private String name = "Anonymous";

   /**
    * Empty no-arg constructor needed by ProActiv
    */
   public Worker() {
   }

   /**
    * Constructor with arguments
```

```
     *
     * @param age
     * @param name
     */
    public Worker(IntWrapper age, String name) {
        this.setAge(age);
        this.setName(name);
    }

    /* (non-Javadoc)
     * @see org.objectweb.proactive.InitActive#initActivity(org.objectweb.proactive.Body)
     */
    public void initActivity(Body body) {
        System.out.println("===> Started Active object: ");
        System.out.println(body.getMBean().getName() + " on " + body.getMBean().getNodeUrl());
    }

    /* (non-Javadoc)
     * @see org.objectweb.proactive.EndActive#endActivity(org.objectweb.proactive.Body)
     */
    public void endActivity(Body body) {
        System.out.println("===> You have killed the active object");
    }
}
```

Here is the skeleton code for a class that can run, suspend, restart and stop a simulation. It uses a implementation of RunActive to provide the necessary control.

```
package org.objectweb.proactive.examples.documentation.classes;

import org.objectweb.proactive.Body;
import org.objectweb.proactive.RunActive;
import org.objectweb.proactive.Service;
import org.objectweb.proactive.api.PALifeCycle;


public class Simulation implements RunActive {
    private boolean stoppedSimulation = false;
    private boolean startedSimulation = false;
    private boolean suspendedSimulation = false;
    private boolean notStarted = true;

    public void startSimulation() {
        // Simulation starts
        System.out.println("Simulation started...");
        notStarted = false;
        startedSimulation = true;
    }

    public void restartSimulation() {
        // Simulation is restarted
        System.out.println("Simulation restarted...");
        startedSimulation = true;
        suspendedSimulation = false;
```

```
    }

    public void suspendSimulation() {
        // Simulation is suspended
        System.out.println("Simulation suspended...");
        suspendedSimulation = true;
        startedSimulation = false;
    }

    public void stopSimulation() {
        // Simulation is stopped
        System.out.println("Simulation stopped...");
        stoppedSimulation = true;
    }

    public void runActivity(Body body) {
        Service service = new Service(body);
        while (body.isActive()) {
            // If the simulation is not yet started wait until startSimulation
            // method
            if (notStarted)
                service.blockingServeOldest("startSimulation");
            // If the simulation is started serve request with FIFO
            if (startedSimulation)
                service.blockingServeOldest();
            // If simulation is suspended wait until restartSimulation method
            if (suspendedSimulation)
                service.blockingServeOldest("restartSimulation");
            // If simulation is stopped, exit
            if (stoppedSimulation) {
                body.terminate();
                PALifeCycle.exitSuccess();
            }
        }
    }
}
```

2.5.5.3. Passing an object implementing the interfaces at creation time

Passing an object implementing the interfaces is the solution to use when we do not control the class that we make active or when you want to write a generic activity policy and reuse it with several active objects. Depending on which phase in the life of the active object we want to customize, we will implement the corresponding interface (one or more) from InitActive , RunActive and EndActive . Following is an example that has a custom activity.

First we need to implement the activity that it will be passed to the active object. We do so by implementing one or more interfaces.

```
package org.objectweb.proactive.examples.documentation.classes;

import org.objectweb.proactive.Body;
import org.objectweb.proactive.RunActive;
import org.objectweb.proactive.Service;


public class LIFOActivity implements RunActive {
```

```
    // -- implements RunActive for serving request in a LIFO order
    public void runActivity(Body body) {
        Service service = new Service(body);
        while (body.isActive()) {
            System.out.println("Serving...");
            service.blockingServeYoungest();
        }
    }
}
```

The implemented interface can be used with any of the creation methods (PAActiveObject.newActive(...), PAActiveObject.newActiveInParallel(...) or PAActiveObject.turnActive(...)). .

```
// Set the constructor parameters
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };

// Instantiate a LIFOActivity object
LIFOActivity activity = new LIFOActivity();

Worker charlie;
try {
    charlie = PAActiveObject.newActive(Worker.class, null, params, null, activity, null);

    System.out.println(charlie.getName() + " is " + charlie.getAge());
    charlie.setAge(new IntWrapper(25));
    charlie.setName("Anonymous");
    System.out.println(charlie.getName() + " is " + charlie.getAge());

    PAActiveObject.terminateActiveObject(charlie, true);
} catch (ActiveObjectCreationException aoExcep) {
    // the creation of ActiveObject failed
    System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
    System.err.println(nodeExcep.getMessage());
}
```

2.5.6. Using the factory pattern with Active Objects

Creating an active object using ProActive might be a little bit cumbersome and requires more lines of code that for creating a regular object. A nice solution to this problem is through the use of the **factory** pattern. This mainly applies to class based creation. It consists in creating a WorkerFactory class with a static method to class that takes care of instantiating the active object and returns it.

```
package org.objectweb.proactive.examples.documentation.classes;

import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.core.node.Node;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.core.util.wrapper.IntWrapper;


/**
 * @author ProActive Team
 *
 * Factory used to instantiate Worker active objects
```

```
 */
public class WorkerFactory {

   /**
    * Create a new Worker active object
    *
    * @param age
    * @param name
    * @param node
    * @return the Worker active object
    */
   public static Worker createActiveWorker(int age, String name, Node node) {
      Object[] params = new Object[] { new IntWrapper(age), name };
      try {
         return (Worker) PAActiveObject.newActive(Worker.class.getName(), params, node);
      } catch (ActiveObjectCreationException e) {
         e.printStackTrace();
         return null;
      } catch (NodeException e) {
         e.printStackTrace();
         return null;
      }
   }
}
```

The static method in the factory class is then used to create active objects:

```
// Creates locally (Node=null) a Worker whose name is Charlie and who is 19
Worker charlie = WorkerFactory.createActiveWorker(19, "Charlie", null);
```

It is up to the programmer to decide whether this method has to throw exceptions or not. We recommend that this method only throw exceptions that appear in the signature of the reified constructor (none here as the constructor of Worker that we call doesn't throw any exception). However, the non functional exceptions induced by the creation of the active object have to be dealt with somewhere in the code.

2.5.7. Using MetaObjectFactory to customize the meta objects

There are many cases where you may want to customize the body used when creating an active object. For instance, one may want to add some debug messages or some timing behavior when sending or receiving requests. The body is a non changeable object that delegates most of its tasks to helper objects called MetaObjects. Standard MetaObjects are already used by default in ProActive but one can easily replace any of those MetaObjects by a custom one.

We have defined a MetaObjectFactory interface that is able to create factories for each of those MetaObjects. This interface is implemented by ProActiveMetaObjectFactory which provides all the default factories used in ProActive.

When creating an active object, it is possible to specify which MetaObjectFactory to use for that particular instance of active object being created.

First you have to write a new MetaObject factory that inherits from ProActiveMetaObjectFactory or directly implements the MetaObjectFactory interface. Inheriting from ProActiveMetaObjectFactory is a great time saver as you only redefine what you really need to as opposed to redefining everything when inheriting from MetaObjectFactory . Here is an example:

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
```

```
import org.objectweb.proactive.core.body.MetaObjectFactory;
import org.objectweb.proactive.core.body.ProActiveMetaObjectFactory;
import org.objectweb.proactive.core.body.UniversalBody;
import org.objectweb.proactive.core.body.request.Request;
import org.objectweb.proactive.core.body.request.RequestFactory;
import org.objectweb.proactive.core.body.request.RequestImpl;
import org.objectweb.proactive.core.mop.MethodCall;


/**
 * @author ProActive Team
 *
 * Customized Meta-Object Factory
 */
public class CustomMetaObjectFactory extends ProActiveMetaObjectFactory {

   private static final MetaObjectFactory instance = new CustomMetaObjectFactory();

   //return a new factory instance
   public static MetaObjectFactory newInstance() {
      return instance;
   }

   private CustomMetaObjectFactory() {
      super();
   }

   protected RequestFactory newRequestFactorySingleton() {
      System.out.println("Creating the  custom metaobject factory...");
      return new CustomRequestFactory();
   }

   protected class CustomRequestFactory extends RequestFactoryImpl implements Serializable {

      public Request newRequest(MethodCall methodCall, UniversalBody sourceBody, boolean isOneWay,
            long sequenceID) {
         System.out.println("Received a new request...");
         return new CustomRequest(methodCall, sourceBody, isOneWay, sequenceID);
      }

      protected class CustomRequest extends RequestImpl {
         public CustomRequest(MethodCall methodCall, UniversalBody sourceBody, boolean isOneWay,
               long sequenceID) {
            super(methodCall, sourceBody, isOneWay, sequenceID);
            System.out.println("I am a custom request handler");
         }
      }
   }
}
```

The factory above simply redefines the RequestFactory in order to make the body use a new type of request. The method protected RequestFactory newRequestFactorySingleton() is one convenience method that ProActiveMetaObjectFactory provides to simplify the creation of factories as singleton.

```
public class ProActiveMetaObjectFactory implements MetaObjectFactory, java.io.Serializable, Cloneable {
    public static final String COMPONENT_PARAMETERS_KEY = "component-parameters";
    public static final String SYNCHRONOUS_COMPOSITE_COMPONENT_KEY = "synchronous-composite";
    protected static Logger logger = ProActiveLogger.getLogger(Loggers.MOP);

    //
    // -- PRIVATE MEMBERS -----------------------------------------------
    //
    // private static final MetaObjectFactory instance = new ProActiveMetaObjectFactory();
    private static MetaObjectFactory instance = new ProActiveMetaObjectFactory();
    public Map<String, Object> parameters = new HashMap<String, Object>();

    //
    // -- PROTECTED MEMBERS -----------------------------------------------
    //
    protected RequestFactory requestFactoryInstance;
    protected ReplyReceiverFactory replyReceiverFactoryInstance;
    protected RequestReceiverFactory requestReceiverFactoryInstance;
    protected RequestQueueFactory requestQueueFactoryInstance;
    protected MigrationManagerFactory migrationManagerFactoryInstance;

    //    protected RemoteBodyFactory remoteBodyFactoryInstance;
    protected ThreadStoreFactory threadStoreFactoryInstance;
    protected ProActiveSPMDGroupManagerFactory proActiveSPMDGroupManagerFactoryInstance;
    protected PAComponentFactory componentFactoryInstance;
    protected ProActiveSecurityManager proActiveSecurityManager;
    protected FTManagerFactory ftmanagerFactoryInstance;
    protected DebuggerFactory debuggerFactoryInstance;
    protected MessageTagsFactory requestTagsFactoryInstance;
    protected Object timItReductor;

    //
    // -- CONSTRUCTORS -----------------------------------------------
    //
    protected ProActiveMetaObjectFactory() {
        this.requestFactoryInstance = newRequestFactorySingleton();
        this.replyReceiverFactoryInstance = newReplyReceiverFactorySingleton();
        this.requestReceiverFactoryInstance = newRequestReceiverFactorySingleton();
        this.requestQueueFactoryInstance = newRequestQueueFactorySingleton();
        this.migrationManagerFactoryInstance = newMigrationManagerFactorySingleton();
        //        this.remoteBodyFactoryInstance = newRemoteBodyFactorySingleton();
        this.threadStoreFactoryInstance = newThreadStoreFactorySingleton();
        this.proActiveSPMDGroupManagerFactoryInstance = newProActiveSPMDGroupManagerFactorySingleton();
        this.ftmanagerFactoryInstance = newFTManagerFactorySingleton();
        this.debuggerFactoryInstance = newDebuggerFactorySingleton();
        this.requestTagsFactoryInstance = newRequestTagsFactorySingleton();
    }

    /**
     * Constructor with parameters
     * It is used for per-active-object configurations of ProActive factories
     * @param parameters the parameters of the factories; these parameters can be of any type
     */
    public ProActiveMetaObjectFactory(Map<String, Object> parameters) {
```

```
        this.parameters = parameters;
        if (parameters.containsKey(COMPONENT_PARAMETERS_KEY)) {
            ComponentParameters initialComponentParameters = (ComponentParameters) parameters
                    .get(COMPONENT_PARAMETERS_KEY);
            this.componentFactoryInstance = newComponentFactorySingleton(initialComponentParameters);
            this.requestFactoryInstance = newRequestFactorySingleton();
            this.replyReceiverFactoryInstance = newReplyReceiverFactorySingleton();
            this.requestReceiverFactoryInstance = newRequestReceiverFactorySingleton();
            this.requestQueueFactoryInstance = newRequestQueueFactorySingleton();
            this.migrationManagerFactoryInstance = newMigrationManagerFactorySingleton();
            //          this.remoteBodyFactoryInstance = newRemoteBodyFactorySingleton();
            this.threadStoreFactoryInstance = newThreadStoreFactorySingleton();
            this.proActiveSPMDGroupManagerFactoryInstance = newProActiveSPMDGroupManagerFactorySingleton();
            this.ftmanagerFactoryInstance = newFTManagerFactorySingleton();
            this.debuggerFactoryInstance = newDebuggerFactorySingleton();
            this.requestTagsFactoryInstance = newRequestTagsFactorySingleton();
        }
    }

    //
    // -- PUBLICS METHODS -----------------------------------------------
    //
    public static MetaObjectFactory newInstance() {
        return instance;
    }

    public static void setNewInstance(MetaObjectFactory mo) {
        instance = mo;
    }

    /**
     * getter for the parameters of the factory (per-active-object config)
     * @return the parameters of the factory
     */
    public Map<String, Object> getParameters() {
        return this.parameters;
    }

    //
    // -- implements MetaObjectFactory -----------------------------------------------
    //
    public RequestFactory newRequestFactory() {
        return this.requestFactoryInstance;
    }

    public ReplyReceiverFactory newReplyReceiverFactory() {
        return this.replyReceiverFactoryInstance;
    }

    public RequestReceiverFactory newRequestReceiverFactory() {
        return this.requestReceiverFactoryInstance;
    }

    public RequestQueueFactory newRequestQueueFactory() {
```

```
    return this.requestQueueFactoryInstance;
}

public MigrationManagerFactory newMigrationManagerFactory() {
    return this.migrationManagerFactoryInstance;
}

//    public RemoteBodyFactory newRemoteBodyFactory() {
//        return this.remoteBodyFactoryInstance;
//    }
public ThreadStoreFactory newThreadStoreFactory() {
    return this.threadStoreFactoryInstance;
}

public ProActiveSPMDGroupManagerFactory newProActiveSPMDGroupManagerFactory() {
    return this.proActiveSPMDGroupManagerFactoryInstance;
}

public PAComponentFactory newComponentFactory() {
    return this.componentFactoryInstance;
}

public FTManagerFactory newFTManagerFactory() {
    return this.ftmanagerFactoryInstance;
}

public DebuggerFactory newDebuggerFactory() {
    return this.debuggerFactoryInstance;
}

public MessageTagsFactory newRequestTagsFactory() {
    return this.requestTagsFactoryInstance;
}

//
// -- PROTECTED METHODS -----------------------------------------------
//
protected RequestFactory newRequestFactorySingleton() {
    return new RequestFactoryImpl();
}

protected ReplyReceiverFactory newReplyReceiverFactorySingleton() {
    return new ReplyReceiverFactoryImpl();
}

protected RequestReceiverFactory newRequestReceiverFactorySingleton() {
    return new RequestReceiverFactoryImpl();
}

protected RequestQueueFactory newRequestQueueFactorySingleton() {
    return new RequestQueueFactoryImpl();
}

protected MigrationManagerFactory newMigrationManagerFactorySingleton() {
```

```
    return new MigrationManagerFactoryImpl();
  }

  //   protected RemoteBodyFactory newRemoteBodyFactorySingleton() {
  //       return new RemoteBodyFactoryImpl();
  //   }
  protected ThreadStoreFactory newThreadStoreFactorySingleton() {
    return new ThreadStoreFactoryImpl();
  }

  protected ProActiveSPMDGroupManagerFactory newProActiveSPMDGroupManagerFactorySingleton() {
    return new ProActiveSPMDGroupManagerFactoryImpl();
  }

  protected PAComponentFactory newComponentFactorySingleton(ComponentParameters
initialComponentParameters) {
    return new ProActiveComponentFactoryImpl(initialComponentParameters);
  }

  protected FTManagerFactory newFTManagerFactorySingleton() {
    return new FTManagerFactoryImpl();
  }

  protected DebuggerFactory newDebuggerFactorySingleton() {
    return new DebuggerFactoryImpl();
  }

  protected MessageTagsFactory newRequestTagsFactorySingleton() {
    return new MessageTagsFactoryImpl();
  }

  // //
  // // -- INNER CLASSES ----------------------------------------------
  // //
  protected static class RequestFactoryImpl implements RequestFactory, java.io.Serializable {
    public Request newRequest(MethodCall methodCall, UniversalBody sourceBody, boolean isOneWay,
        long sequenceID, MessageTags tags) {
      //############# exemple de code pour les nouvelles factories
      //          if(System.getProperty("migration.stategy").equals("locationserver")){
      //              return new RequestWithLocationServer(methodCall, sourceBody,
      //             isOneWay, sequenceID, LocationServerFactory.getLocationServer());
      //          }else{
      return new org.objectweb.proactive.core.body.request.RequestImpl(methodCall, sourceBody,
        isOneWay, sequenceID, tags);
      //}
    }
  }

  // end inner class RequestFactoryImpl
  protected static class ReplyReceiverFactoryImpl implements ReplyReceiverFactory, java.io.Serializable {
    public ReplyReceiver newReplyReceiver() {
      return new org.objectweb.proactive.core.body.reply.ReplyReceiverImpl();
    }
  }
```

```
  // end inner class ReplyReceiverFactoryImpl
  protected class RequestReceiverFactoryImpl implements RequestReceiverFactory, java.io.Serializable {
      public RequestReceiver newRequestReceiver() {
          if
(ProActiveMetaObjectFactory.this.parameters.containsKey(SYNCHRONOUS_COMPOSITE_COMPONENT_KEY) &&
              ((Boolean) ProActiveMetaObjectFactory.this.parameters
                      .get(ProActiveMetaObjectFactory.SYNCHRONOUS_COMPOSITE_COMPONENT_KEY)).booleanValue())
{
              return new SynchronousComponentRequestReceiver();
          }
          return new org.objectweb.proactive.core.body.request.RequestReceiverImpl();
      }
  }

  // end inner class RequestReceiverFactoryImpl
  protected class RequestQueueFactoryImpl implements RequestQueueFactory, java.io.Serializable {
      public BlockingRequestQueue newRequestQueue(UniqueID ownerID) {
          if ("true".equals(ProActiveMetaObjectFactory.this.parameters
                  .get(SYNCHRONOUS_COMPOSITE_COMPONENT_KEY))) {
              return null;
          }

          //if (componentFactoryInstance != null) {
          // COMPONENTS
          // we need a request queue for components
          //return new ComponentRequestQueueImpl(ownerID);
          //} else {
          return new org.objectweb.proactive.core.body.request.BlockingRequestQueueImpl(ownerID);
          //}
      }
  }

  // end inner class RequestQueueFactoryImpl
  protected static class MigrationManagerFactoryImpl implements MigrationManagerFactory,
          java.io.Serializable {
      public MigrationManager newMigrationManager() {
          //############# example de code pour les nouvelles factories
          //          if(System.getProperty("migration.stategy").equals("locationserver")){
          //              return new MigrationManagerWithLocationServer(LocationServerFactory.getLocationServer());
          //          }else{
          return new org.objectweb.proactive.core.body.migration.MigrationManagerImpl();
          //}
      }
  }


  // end inner class RemoteBodyFactoryImpl
  protected static class ThreadStoreFactoryImpl implements ThreadStoreFactory, java.io.Serializable {
      public ThreadStore newThreadStore() {
          return new org.objectweb.proactive.core.util.ThreadStoreImpl();
      }
  }
```

```
  // end inner class ThreadStoreFactoryImpl
  protected static class ProActiveSPMDGroupManagerFactoryImpl implements
ProActiveSPMDGroupManagerFactory,
        java.io.Serializable {
    public ProActiveSPMDGroupManager newProActiveSPMDGroupManager() {
      return new ProActiveSPMDGroupManager();
    }
  }

  // end inner class ProActiveGroupManagerFactoryImpl
  protected class ProActiveComponentFactoryImpl implements PAComponentFactory, java.io.Serializable {
    // COMPONENTS
    private ComponentParameters componentParameters;

    public ProActiveComponentFactoryImpl(ComponentParameters initialComponentParameters) {
      this.componentParameters = initialComponentParameters;
    }

    public PAComponent newPAComponent(Body myBody) {
      return new PAComponentImpl(this.componentParameters, myBody);
    }
  }

  // FAULT-TOLERANCE
  protected class FTManagerFactoryImpl implements FTManagerFactory, Serializable {
    public FTManager newFTManager(int protocolSelector) {
      switch (protocolSelector) {
        case FTManagerFactory.PROTO_CIC_ID:
          return new FTManagerCIC();
        case FTManagerFactory.PROTO_PML_ID:
          return new FTManagerPMLRB();
        default:
          logger.error("Error while creating fault-tolerance manager : " +
            "no protocol is associated to selector value " + protocolSelector);
          return null;
      }
    }

    public FTManager newHalfFTManager(int protocolSelector) {
      switch (protocolSelector) {
        case FTManagerFactory.PROTO_CIC_ID:
          return new HalfFTManagerCIC();
        case FTManagerFactory.PROTO_PML_ID:
          return new HalfFTManagerPMLRB();
        default:
          logger.error("Error while creating fault-tolerance manager : " +
            "no protocol is associated to selector value " + protocolSelector);
          return null;
      }
    }
  }

  protected static class DebuggerFactoryImpl implements DebuggerFactory, java.io.Serializable {
    public Debugger newDebugger() {
```

```
        return new DebuggerImpl();
    }
  }

  // REQUEST-TAGS
  protected static class MessageTagsFactoryImpl implements MessageTagsFactory, Serializable {

    /**
     * @see MessageTagsFactory#newMessageTags()
     */
    public MessageTags newMessageTags() {
        return new MessageTags();
    }
  }

  // SECURITY
  public void setProActiveSecurityManager(ProActiveSecurityManager psm) {
    this.proActiveSecurityManager = psm;
  }

  public ProActiveSecurityManager getProActiveSecurityManager() {
    return this.proActiveSecurityManager;
  }

  @Override
  public Object clone() throws CloneNotSupportedException {

    try {
        return MakeDeepCopy.WithObjectStream.makeDeepCopy(this);
    } catch (IOException e) {
        //TODO replace by CloneNotSupportedException(Throwable e) java 1.6
        throw (CloneNotSupportedException) new CloneNotSupportedException(e.getMessage()).initCause(e);
    } catch (ClassNotFoundException e) {
        throw (CloneNotSupportedException) new CloneNotSupportedException(e.getMessage()).initCause(e);
    }
  }

  public void setTimItReductor(Object timItReductor) {
    this.timItReductor = timItReductor;
  }

  public Object getTimItReductor() {
    return this.timItReductor;
  }

}
```

More explanations can be found in the javadoc [3] of the org.objectweb.proactive.core.body.ProActiveMetaObjectFactory class. The use of that factory is fairly simple. All you have to do is to pass an instance of the factory when creating a new active object. If we take the same example as before we have:

```
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };
```

[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../api_published/index.html

```
try {
    Worker charlie = (Worker) PAActiveObject.newActive(Worker.class.getName(), null, params, null,
            null, CustomMetaObjectFactory.newInstance());
} catch (ActiveObjectCreationException e) {
    e.printStackTrace();
} catch (NodeException e) {
    e.printStackTrace();
}
```

2.6. Elements of an active object and futures

In this section, we'll have a very close look at what happens when an active object is created. This section aims at providing a better understanding of how the library works and where the restrictions of Proactive come from.

Consider that some code of a class A creates an active object of class Worker using a piece of code like this:

```
// Set the constructor parameters
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };

Worker charlie;
try {
    charlie = PAActiveObject.newActive(Worker.class, params);
} catch (ActiveObjectCreationException aoExcep) {
    // the creation of ActiveObject failed
    System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
    System.err.println(nodeExcep.getMessage());
}
```

If the creation of the active instance of A is successful, the graph of objects is as described in the following figure (with arrows denoting references):

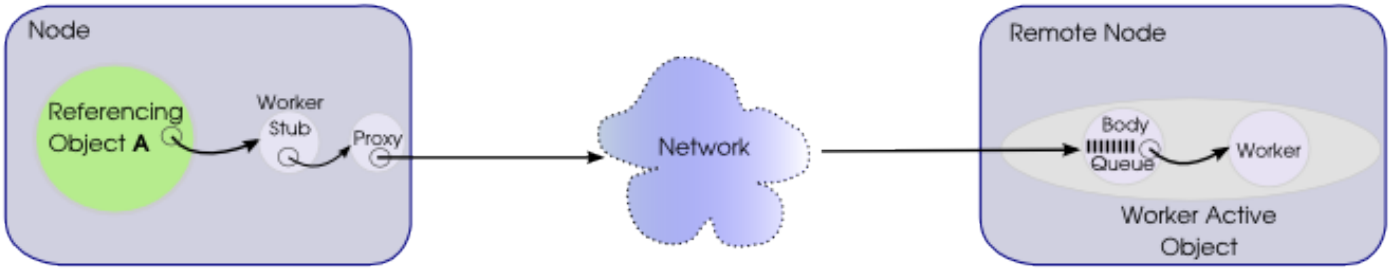


Figure 2.2. The components of an active object and of a referencing object

The active instance of Worker is composed of 2 objects:

- a body (Body)
- an instance of Worker

Besides the active object composed of a body and an object, there is also a proxy and a stub for each active or passive object referencing the active object.

2.6.1. Role of the stub

The role of the class Stub_Worker is to reify all method calls that can be performed through a reference of type Worker. Reifying a call simply means constructing an object that represents the call (in our case, all reified calls are instance of class org.objectweb.proactive.core.mop.MethodCall), so it can be manipulated as any other object. The reified call is then processed by the other components of the active object in order to achieve the behavior we expect from an active object.

Using a standard object for representing elements of the language that are not normally objects (such as method calls, constructor calls, references, types,...) is at the core of **metaobject programming**. The ProActive metaobject protocol (MOP) is is described in Chapter 43. MOP: Metaobject Protocol[4]. but it is not a prerequisite for understanding and using ProActive.

As one of our objectives is to provide transparent active objects, references to active objects of class Worker need to be of the same type as references to passive instances of Worker (this feature is called **polymorphism** between passive and active instances of the same class). This is why Stub_Worker is a subclass of class Worker by construction, therefore allowing instances of class Stub_Worker to be assigned to variables of type Worker.

Class Stub_Worker redefines each of the methods inherited from its superclasses. The code of each method of class Stub_Worker builds an instance of class org.objectweb.proactive.core.mop.MethodCall in order to represent the call to this method. This object is then passed to the BodyProxy, which returns an object that is returned as the result of the method call. From the caller's point of view, everything looks like if the call had been performed on an instance of Worker.

Now that we know how stubs work, we can understand some of the limitations of ProActive:

- Obviously, Stub_Worker cannot redefine final methods inherited from class Worker. If methods are final, calls to these methods are not reified but are executed on the stub, which may lead to unpredictable behavior.

  As there are 6 final methods in the base class Object, there is the question of how ProActive deals with these methods. In fact, 5 out of this 6 methods deal with thread synchronization (notify(), notifyAll() and the 3 versions of wait()). Those methods should not be used since an active object provides thread synchronization. Using the standard thread synchronization mechanism and ProActive thread synchronization mechanism at the same time might conflict and result in an very hard to debug behaviour.

  The last final method in the class Object is getClass(). When invoked on an active object, getClass() is not reified and therefore performed on the stub object, which returns an object of class Class that represents the class of the stub (Stub_Worker in our example) and not the class of the active object itself (Worker in our example). However, this method is seldom used in standard applications and it doesn't prevent the operator instanceof from working thanks to its polymorphic behavior. Therefore the expression (foo instanceof Worker) has the same value whether Worker is active or not.
- Getting or setting instance variables directly (not through a get or a set method) must be avoided for active objects because it results in getting or setting the value on the stub object and not on the instance of the class Worker. This problem is usually worked around by using get/set methods for setting or reading attributes. This rule of strict encapsulation can also be found in JavaBeans or in most distributed object systems like RMI or CORBA.

2.6.2. Role of the proxy

The role of the proxy is to handle asynchronism in calls to active object. More specifically, it creates future objects if possible and needed, forwards calls to bodies, and returns future objects to the stubs. As this class operates on MethodCall objects, it is absolutely generic and does not depend at all on the type of the stub that feeds calls in through its reify method.

2.6.3. Role of the body

The body is responsible for storing calls (actually, Request objects) in a queue of pending requests and processing these requests according to a given synchronization policy. The default behaviour for the synchronization policy is FIFO. The Body has its own thread which chooses requests from the queue of requests and executes the associated call.

[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/MOP.html

2.6.4. Role of the instance of class Worker

There is also a standard instance of class Worker. It may contain some synchronized information in its live method, if any. As the body executes calls one by one, there cannot be any concurrent execution of two portions of code for this object by two different threads. This enables the use of pre and post-conditions and class invariants. As a consequence, the use of the keyword synchronized in class Worker should not be necessary. Any synchronization scheme that can be expressed through monitors and synchronized statements can be expressed using ProActive's high-level synchronization mechanism in a much more natural and user-friendly way.

2.7. Asynchronous calls and futures

2.7.1. Creation of a Future Object

Whenever it is possible, a method call on an active object is reified as an asynchronous request. In that case, it immediately returns a future object. If it is not possible, the call is synchronous and blocks until the reply is received.

This future object acts as a placeholder for the result of the not-yet-performed method invocation. As a consequence, the calling thread can go on with executing its code, as long as it does not need to invoke methods on the returned object. At the time when the real result is needed, the calling thread is automatically blocked if the result of the method invocation is not yet available. Below are shown the different cases that can lead to an asynchronous call. Note that this table does not apply when using the tryWithCatch annotators to deal with asynchronous exceptions. See Chapter 5, *Exception Handling*.

Method Signature		**Creation of a future**	**Asynchronous**
Return type	**Can throw checked exception**		
Void	No	No	Yes
Void	Yes	No	No
Non Reifiable Object	No	No	No
Non Reifiable Object	Yes	No	No
Reifiable Object	No	Yes	Yes
Reifiable Object	Yes	No	No
Array	No	No	No
Array	yes	No	No

Table 2.1. Future creation, and asynchronous calls depending on return type

As we can see, the creation of a future depends not only on the caller type, but also on the return object type. Creating a future is only possible if the object is reifiable. Although a future has a similar structure to an active object, a future object is not active. It only has a Stub and a Proxy as shown in the figure below:

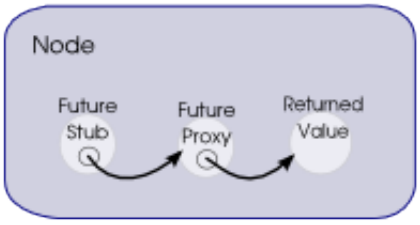


Figure 2.3. A future object

During its lifetime, an active object can create many future objects. There are all automatically kept in a FuturePool.

Each time a future is created, it is inserted in the future pool of the corresponding active object. When the result becomes available, the future object is removed from the pool. There are several methods in org.objectweb.proactive.api.ProFuture which provide control over the way futures are returned.

2.7.2. Methods affected by futures

Any call to a future object is reified in order to be blocked if the future is not yet available and later executed on the result object. However, two methods don't follow this scheme: equals and hashCode. They are often called by other methods from the Java library, like HashTable.add() and so are most of the time out of control from the user. This can lead very easily to deadlocks if they are called on a not yet available object.

There are some drawbacks with this technique, the main one being the impossibility to have a user override of the default HashTable and equals() methods.

- **hashCode()** - Instead of returning the hashcode of the object, it returns the hashcode of its proxy. Since there is only one proxy per future object, there is a unique equivalence between them.
- **equals()** - The default implementation of equals() in the Object class is to compare the references of two objects. In ProActive, it is redefined to compare the hashcode of two proxies. As a consequence, it is only possible to compare two future object, and not a future object with a normal object.
- **toString()** - The toString() method is most of the time called with System.out.println() to turn an object into a printable string. In the current implementation, a call to this method will block on a future object like any other call, thus, one has to be careful when using it. As an example, trying to print a future object for debugging purpose will most of the time lead to a deadlock. Instead of displaying the corresponding string of a future object, you might consider displaying its hashCode.

2.7.3. Asynchronous calls in details

2.7.3.1. The setup

First, let us introduce the example we will use throughout this section. Let us say that some piece of code of a class (that we named Caller) calls the method foo() on an active instance of class Worker. This call is asynchronous and returns a future object of class Value. Then, possibly after having executed some other code, the same thread that issued the call calls the method bar() on the future object returned by the call to foo().

The code for the following example is:

```
package org.objectweb.proactive.examples.documentation.activeobjectconcepts;

import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.core.util.wrapper.IntWrapper;
import org.objectweb.proactive.examples.documentation.classes.Value;
import org.objectweb.proactive.examples.documentation.classes.Worker;


public class Caller {

  public static void asynchronousCall() {
    try {

      Object[] params = new Object[] { new IntWrapper(26), "Charlie" };
      Worker charlie = PAActiveObject.newActive(Worker.class, params);
      Value v = charlie.foo();
      v.bar();

    } catch (ActiveObjectCreationException aoExcep) {
```

```
            System.err.println(aoExcep.getMessage());
        } catch (NodeException nodeExcep) {
            System.err.println(nodeExcep.getMessage());
        }
    }

    public static void main(String[] args) {
        asynchronousCall();
    }
}
```

2.7.3.2. What would have happened in a sequential world

In a sequential world, the method in charge of the call would look like as follows:

```
public static void synchronousCall() {
    Worker charlie = new Worker(new IntWrapper(26), "Charlie");
    Value v = charlie.foo();
    v.bar();
}
```

In this single-threaded version, the thread executes the code of the caller object Caller up to the call of foo. Then, the code of foo() in class Worker is executed. After that, the thread comes back to the code of the calling method in object Caller up to the call to bar(). Next, the code of bar() in class Value is executed and finally, the thread returns to the code of the calling method in class Caller until its end. The sequence diagram below summarizes this execution. You can notice how the single thread successively executes code of different methods in different classes.

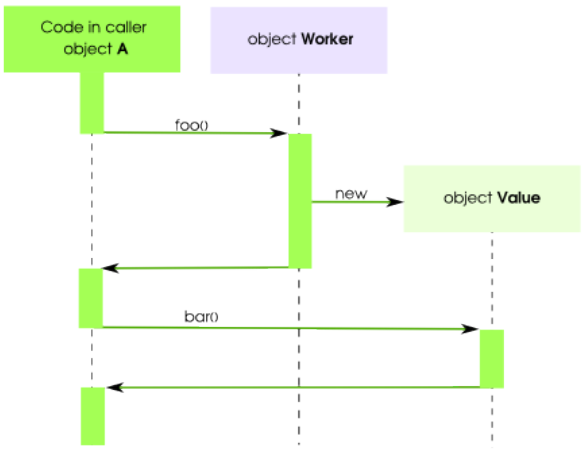


Figure 2.4. Single-threaded version of the program

2.7.3.3. Visualizing the graph of active and passive objects

Now, we will show how an active object handles the call for the code above. Instead of blocking on the execution of foo(), it will create a future and continue execution. Let us first get an idea of what the graph of objects at execution (the objects with their references to each other) looks like at three different moments of the execution:

- Before calling foo(), we have exactly the same setup as after the creation of the active instance of Worker: an instance of class Caller and an active instance of class Worker. As with all active objects, the instance of the class Worker is composed a Body with a request queue and the actual instance of Worker.

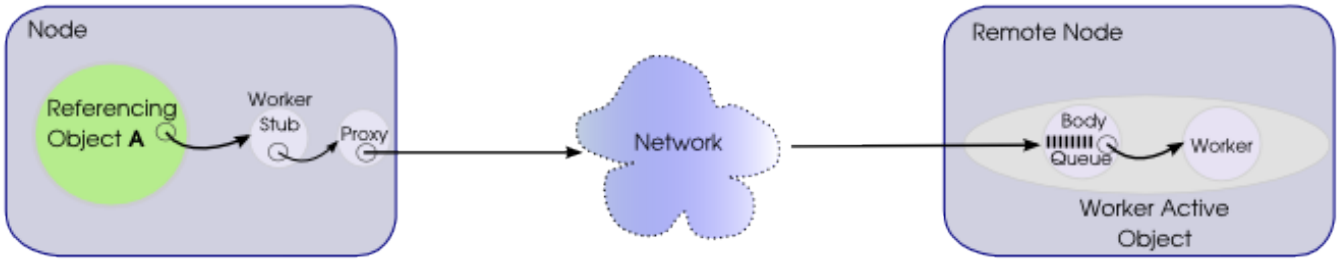


Figure 2.5. The components of an active object and of a referencing object

- After the asynchronous call of foo(), Caller now holds a reference onto a future object representing the not yet available result of the call. It is actually composed of a Stub_Value and a FutureProxy as shown on the figure below.

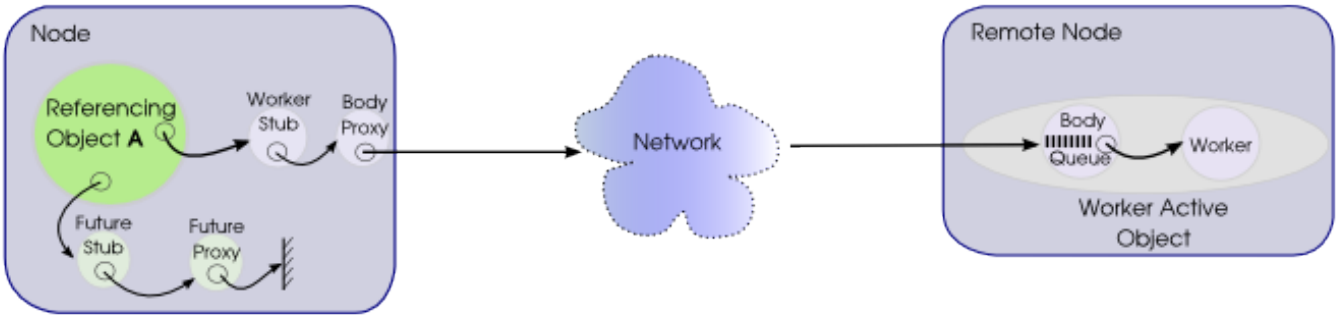


Figure 2.6. The components of a future object before the result is set

- Right after having executed foo() on the instance of Worker, the thread of the Body sets the result in the future. For that, it gives FutureProxy a reference onto a Value (see figure below).

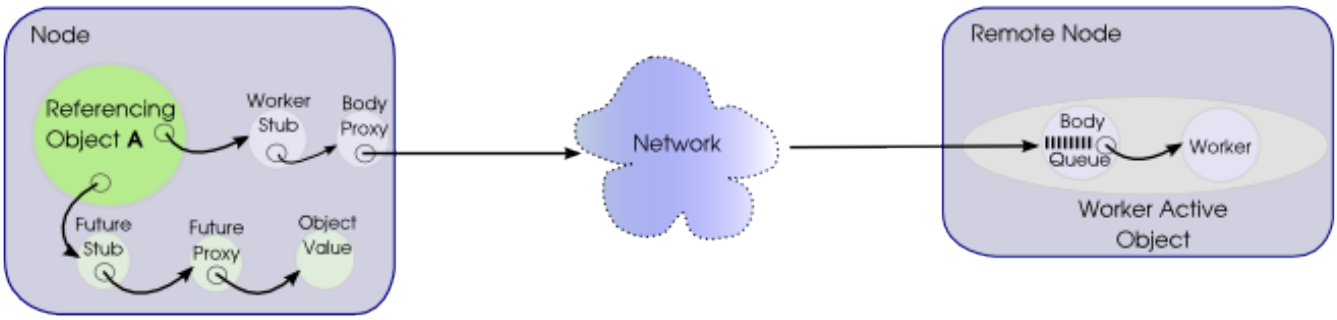


Figure 2.7. All components of a future object

2.7.3.4. Sequence Diagram

The sequence of calls is more complex for the active object case. We have now two threads: the thread that belongs to the subsystem of Caller and the one that belongs to the subsystem of Worker. We will call these the first and second threads.

The first thread invokes foo() on an instance of Stub_Worker, which builds a MethodCall object and passes it to the BodyProxy as a parameter of the call to be reified. The proxy then checks the return type of the call (in this case Value) and generates a future object of type Value for representing the result of the method invocation. The future object is actually composed of a Stub_V and a FutureProxy. A reference onto this future object is set in the MethodCall object. This reference will be used to set the value after the call has been executed. When the MethodCall object is ready, it is passed as a parameter for a Request to the Body of the Worker active object. The Body simply appends this request to the queue of pending requests and returns immediately. The call to foo() issued by Caller returns a future object of type Stub_Value, that is a subclass of Value.

At some point, possibly after having served some other requests, the second thread (the active object's thread) picks up the previous request issued by the first thread. It then executes the embedded call by calling foo() on the instance of Worker with the actual parameters stored in the MethodCall object. As specified in its signature, this call returns an object of type Value. The second thread is responsible for setting this object in the future object (which is the reason why MethodCall objects hold a reference onto the future object created by the FutureProxy). The execution of the call is now over, and the second thread can select and serve another request from the queue.

In the meantime, the first thread has continued executing the code of the calling method in class Caller. At some point, it calls bar() on the object of type Stub_Value that was returned by the call of foo(). This call is reified thanks to the Stub_Value and processed by the FutureProxy. If the object the future represents is available (the second thread has already set it in the future object), the call is executed and returns a value to the calling code in Caller. The sequence diagram below presents the process graphically.

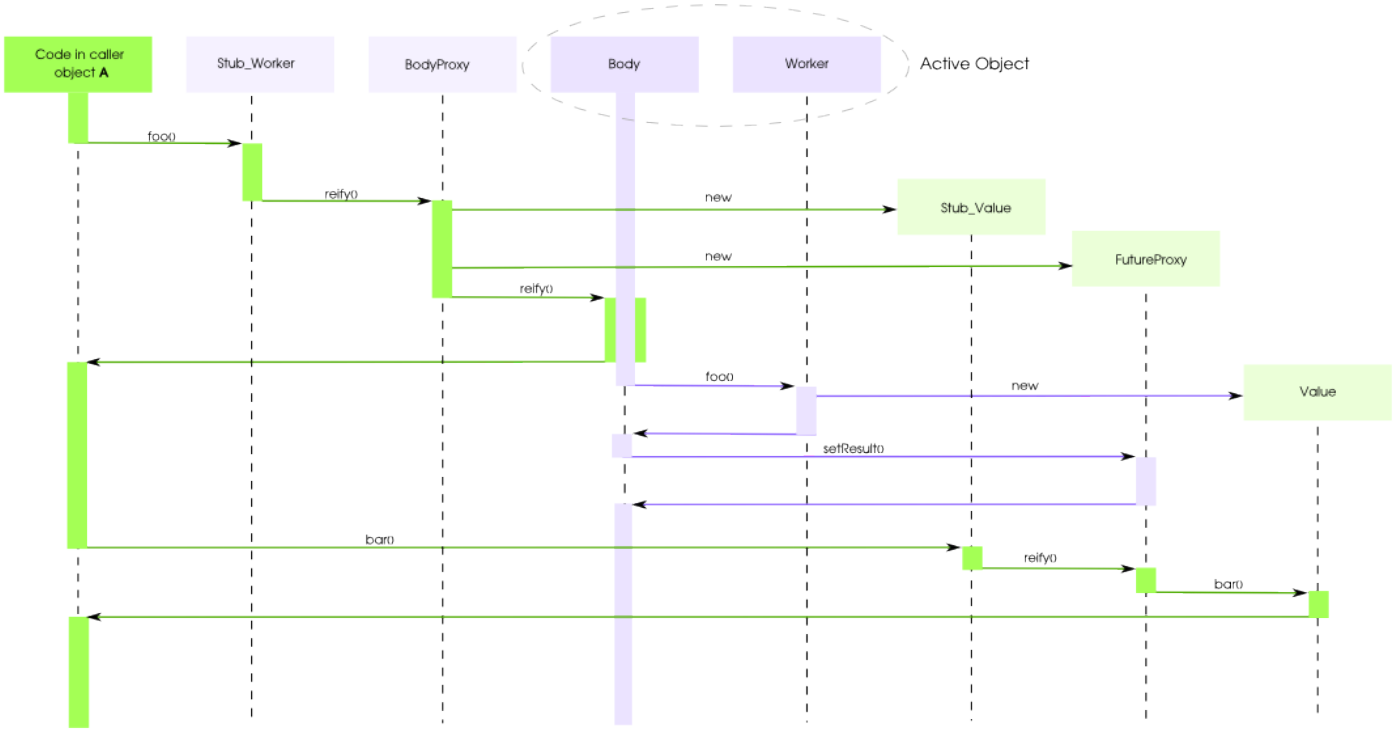


Figure 2.8. Call flow when a future is set before being used

If the value in the future it is not available yet, the first thread is suspended in FutureProxy until the second thread sets the result in the future object (see figure below). This is called wait-by-necessity.

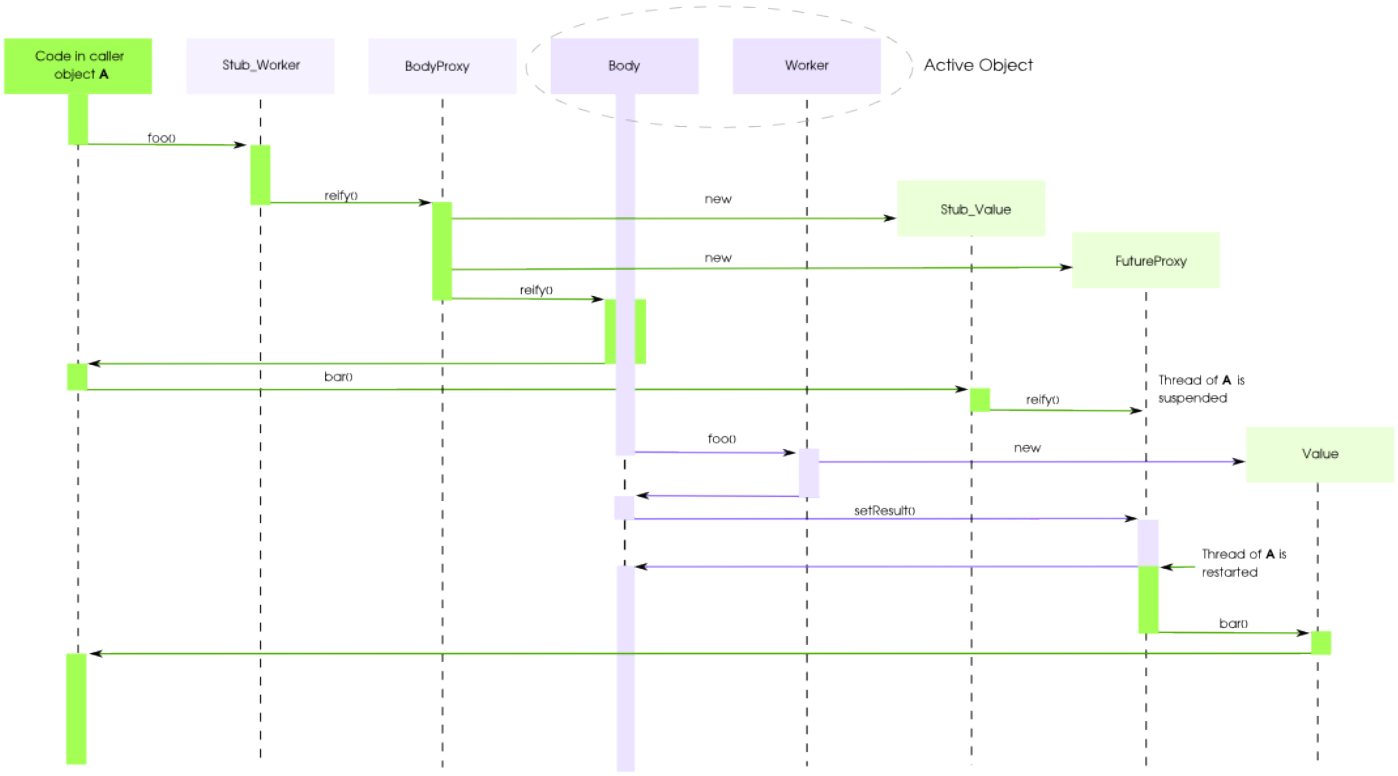


Figure 2.9. Call flow when the future is not set before being used (wait-by-necessity)

2.7.4. Good ProActive programming practices

There are few things to remember with active objects, futures, and asynchronous method calls, in order to avoid annoying debugging sessions:

- **A constructor with no-args** needs to be used either for the Active Objects creation (if not present, an exception might be thrown) or Future creation for a method call (if not present, the method call is synchronous). Avoid placing initialization code in this constructor, as it may lead to unexpected behavior because this constructor is called for the stub creation.
- Make your classes implement **Serializable** interface since ProActive deals with objects that will be sent across networks.
- Use **wrappers** instead of primitive types or final classes for methods result types. Otherwise, you will loose the asynchronism capabilities. For instance, if one of your object has a method whose signature looks like this:

  ```
  int giveSolution(parameter)
  ```

  calling this method with ProActive is synchronous. To have asynchronous calls, use:

  ```
  IntWrapper giveSolution(parameter)
  ```

  All wrappers are in the package: `org.objectweb.proactive.core.util.wrapper`

  ProActive provides several primitive type wrappers, with 2 versions of each, one **mutable**, and another **immutable**.

  To make the call asynchronous, only the methods return types have to use wrappers. The parameters can use the regular Java types.
- Avoid to return `null` in Active Object methods: on the caller side the test

  ```
  if(result_from_method == null)
  ```

has no sense. Result_from_method is a couple Stub - FutureProxy as explained above, so even if the method returns null, result_from_method cannot be null:

```
public Worker getWorker() {
   if (this.name.compareTo("Anonymous") != 0) {
      return this;
   } else {
      return null; //to avoid in ProActive
   }
}
```

On the caller side:

```
Worker charlie = (Worker) PAActiveObject.newActive(Worker.class.getName(), null);
Worker worker = charlie.getWorker();
if (worker == null) {
   System.out.println("worker is null");
   worker.display();
} else {
   System.out.println("worker is not null");
   worker.display();
}
```

This test is never true because either worker is equal to Stub-->Proxy-->null if the future is not available yet or if the method returns null, or worker is equal to Stub-->Proxy-->Object if the future is available. But worker is never null. In our case, the method has returned null since worker's name is Anonymous and the execution trace of the caller is the following:

```
            === Started Active object:
            org.objectweb.proactive.examples.documentation.classes.Worker on rmi://kisscool.inria.fr:6618/
Node1346108379
            worker is not null
            Exception in thread "main" java.lang.NullPointerException
              at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
              at sun.reflect.NativeMethodAcccumentation.activeobjectconcepts.Caller.goodPractice(Caller.java:71)
              at org.objectweb.proactive.examples.documentation.activeobjectconcepts.Caller.main(Caller.java:86)
```

As you can see, worker is considered as not null and, as we call one of its method and since it is actually null, we get a NullPointerException.

- While using synchronous calls within your code is discouraged, it happens that sometimes you cannot avoid using them. To mimic the behaviour of a synchronous call on top of our asynchronous framework, synchronous calls are blocked within the ProActive Programming layer until the result is computed and the future filled with the result. If the future is not filled for any reason, the call is going to be blocked for ever and there is nothing you can do. To address this issue, it is possible to set a timeout for this kind of method call. The property to modify is proactive.future.synchrequest.timeout. By default, the value is 0, meaning an unlimited timeout. When this value is set to a positive long value, this value is used as timeout value. When the timeout is reached, a ProActiveTimeoutException is thrown. To catch the exception, just enclosed the synchronous method call within a try/catch block and define the appropriate treatment in the catch block.

2.7.5. Lightweight failure detection

Waiting for the update of a future in a ProActive application could freeze the application if the node responsible for updating the future experiences a failure. To prevent this, ProActive has a comprehensive generic fault tolerance mechanism. However, it can be too expensive to checkpoint all the active objects, when fault detection is only needed for certain objects.

By default, ProActive continuously pings active objects awaited for updating futures. When such a ping fails, the associated awaited future is updated with a runtime exception. Accessing the future will throw an exception.

```
Worker ao = PAActiveObject.newActive(Worker.class, null);
IntWrapper future = ao.getAge();
String str;
try {
   str = future.toString();
} catch (FutureMonitoringPingFailureException fmpfe) {
   System.out.println("The active object 'ao' had a failure");
   fmpfe.printStackTrace();
}
```

The ping is started when the future is being awaited, but it is also possible to start it beforehand using the ProFuture.monitorFuture(java.lang.Object future) method. To find out more about the fault tolerance mechanism read Chapter 32. Fault-Tolerance[5].

2.8. Automatic Continuation in ProActive

Automatic continuation is the propagation of a future outside the activity that has sent the corresponding request.

Automatic continuations allows using non-updated futures in the normal program flow without blocking to wait for updating the result contained in the future. When the result is available on the object that originated the creation of the future, this object updates the result in all the objects to which it passed the future. If the objects passed the future further they will updated the value in the corresponding objects.

2.8.1. Sending Futures

An automatic continuation can occur either when sending a request (request parameter is or contains a future) or when sending a reply (the result is or contains a future). Outgoing futures are registered in the FuturePool of the active object which sends the future. For instance, In the example presented in the Section 2.8.3, “Illustration of an Automatic Continuation”, a FuturePool is created on the Caller side. This FuturePool is a pool of couples <Future, Destination> and is filled in when the future is serialized. Every request or reply are serialized before being sent, and the future is part of the request or the reply. A thread Thread sending the message keeps in a static table (FuturePool.bodiesDestination) a reference to the destination body. Hence, when a future Future is serialized by the same thread, this thread looks up in the static table the destination Destination registered for the thread Thread. If it finds a destination for the future, the future notifies the FuturePool that it is going to leave, which in turn registers the <Future, Destination> as an automatic continuation.

When the value is available for the future, it is is propagated to all objects that received the future Future. This type of update is realized by a thread located in the FuturePool.

2.8.2. Receiving Futures

When a message containing a future is received by an active object the active object registers the future in the **FuturePool** to be able to update it when the value will be available. After the future is deserialized, it is registered in a static table FuturePool.incomingFutures.

2.8.3. Illustration of an Automatic Continuation

[5] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/faultTolerance.html

To illustrate how automatic continuation takes place, we will use three classes: Caller, Value and Worker. The Caller and Value classes are used to instantiate passive objects and the Worker class is used to instantiate an active object. The Worker class has a foo() method and the Value class has a bar(Value v, int nbYears) method.

```
Value v = new Value();
Worker worker = PAActiveObject.newActive(Worker.class, null);
Value v1 = worker.foo(); //v1 is a future
Value v2 = v.bar(v1, 1); //v1 is passed as parameter
```

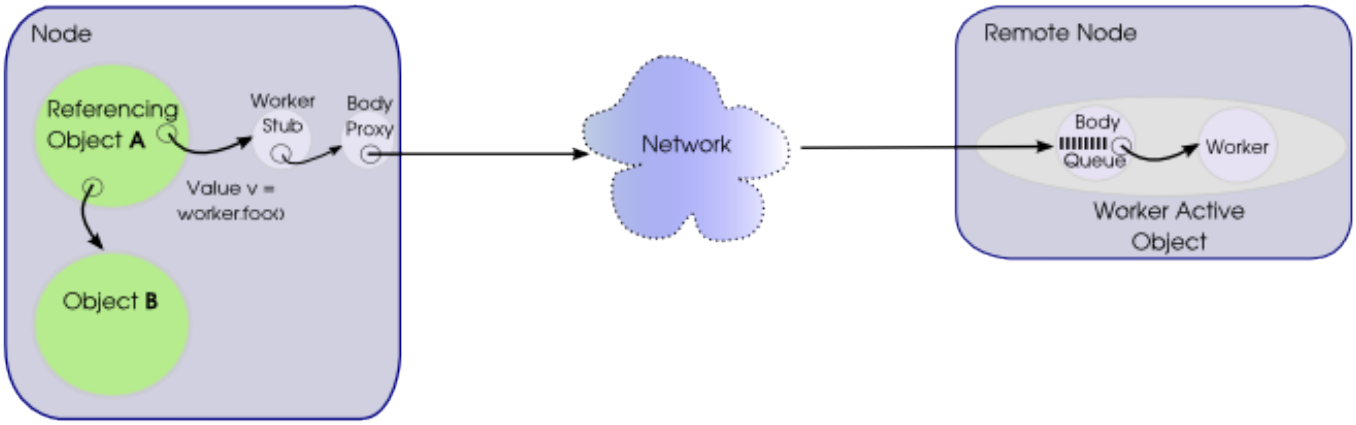
where

```
public Value foo() {
   return new Value(this.age, this.name);
}
```

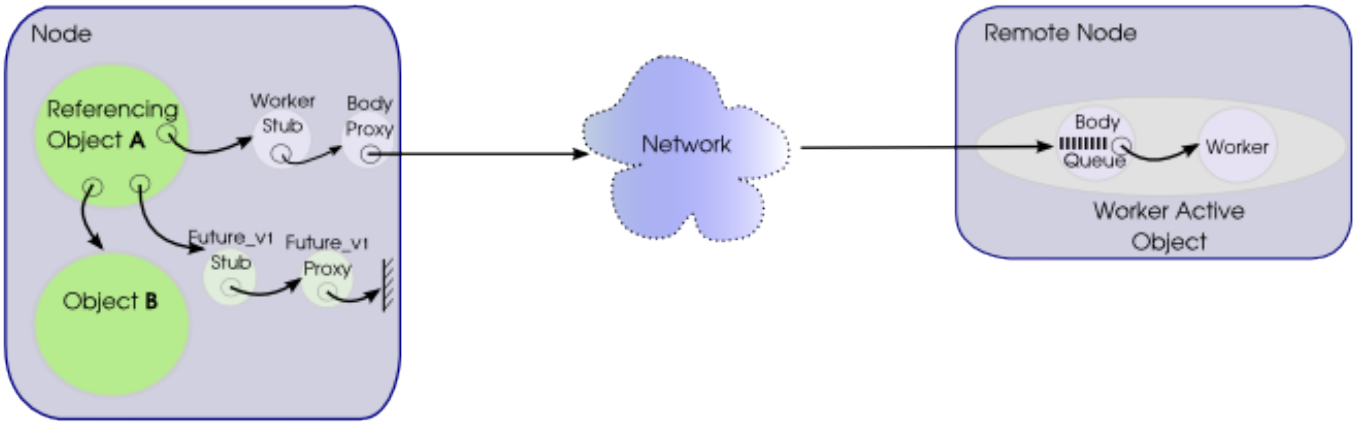
and

```
public Value bar(Value v, int nbYears) {
   v.setAge(new IntWrapper(v.getAge().getIntValue() + nbYears));
   return this;
}
```

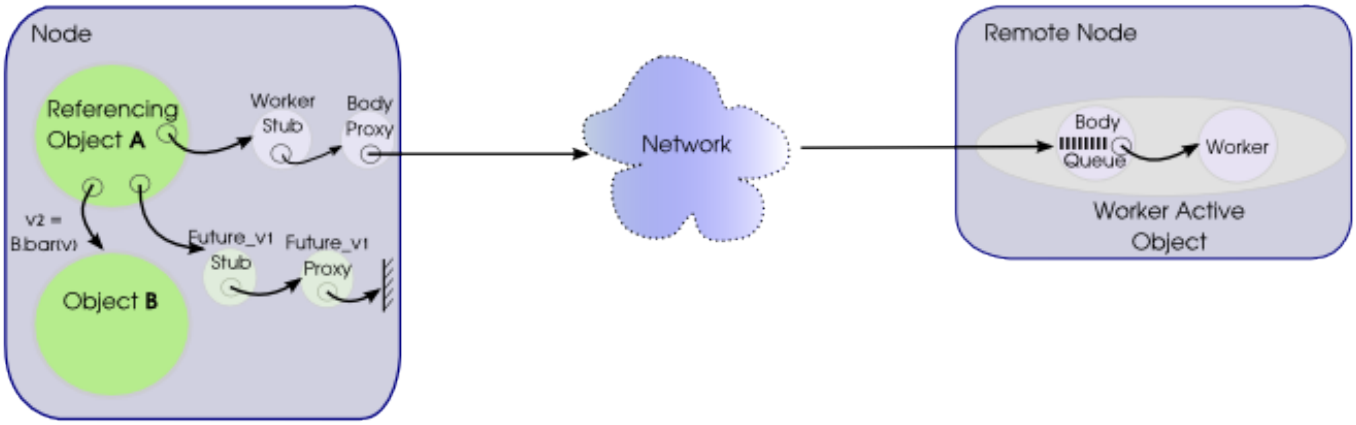
We will first call foo() on the worker. The call goes through the worker stub and the body proxy and it is placed in the queue of the body. We are using a passive object as an example, however the process would be similar if the caller was an active object or if the objects were all on the same machine. In those cases, the call will also go through the stub and proxy as described in the sequence diagrams above. The following pictures represent the continuation process. Note that in this diagram, the caller is named A and the value is named B.



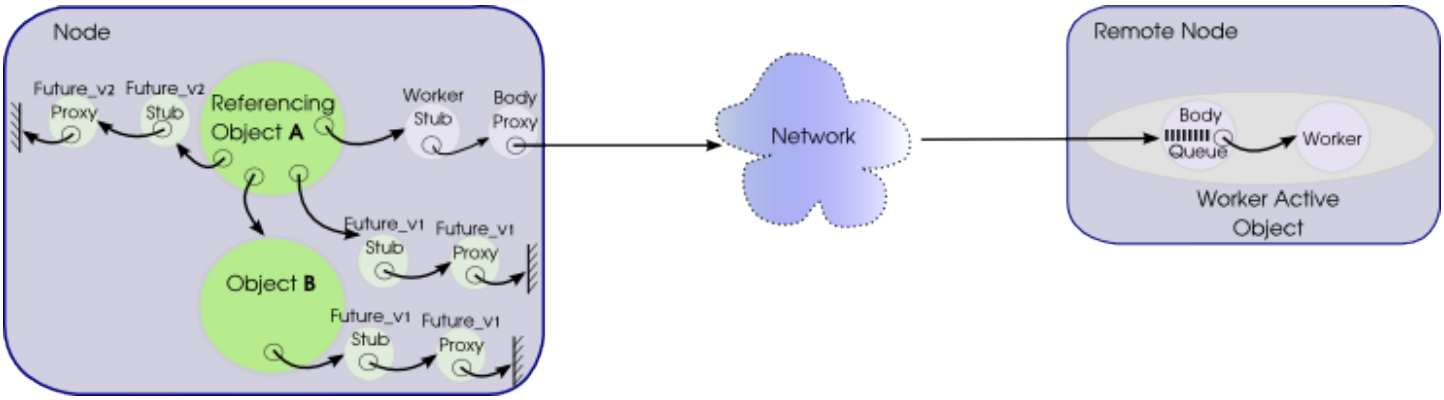
After sending the request, the thread of object A continues immediately (asynchronous call) and v1 points to a future that has not been updated yet.



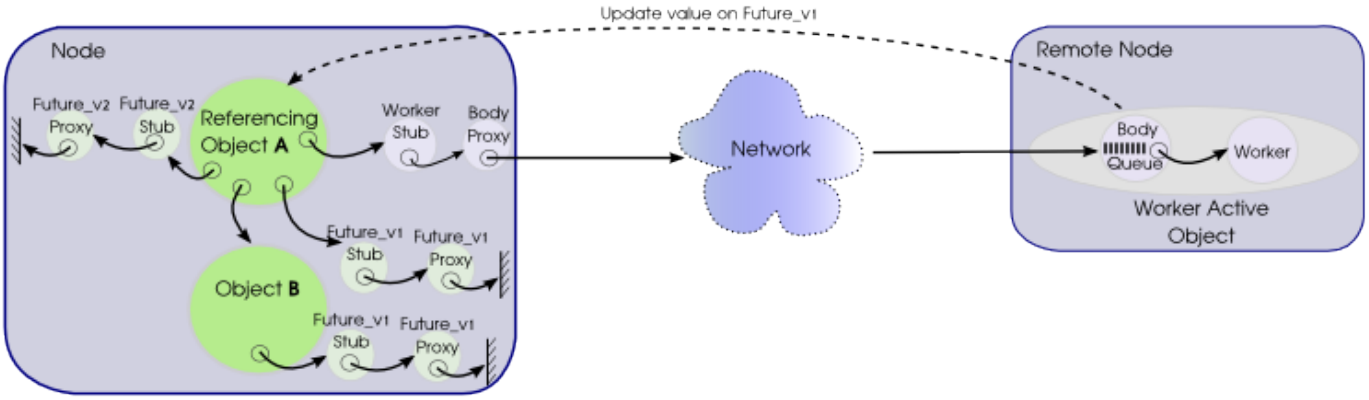
Next, object A calls bar(v1, 1) on B.



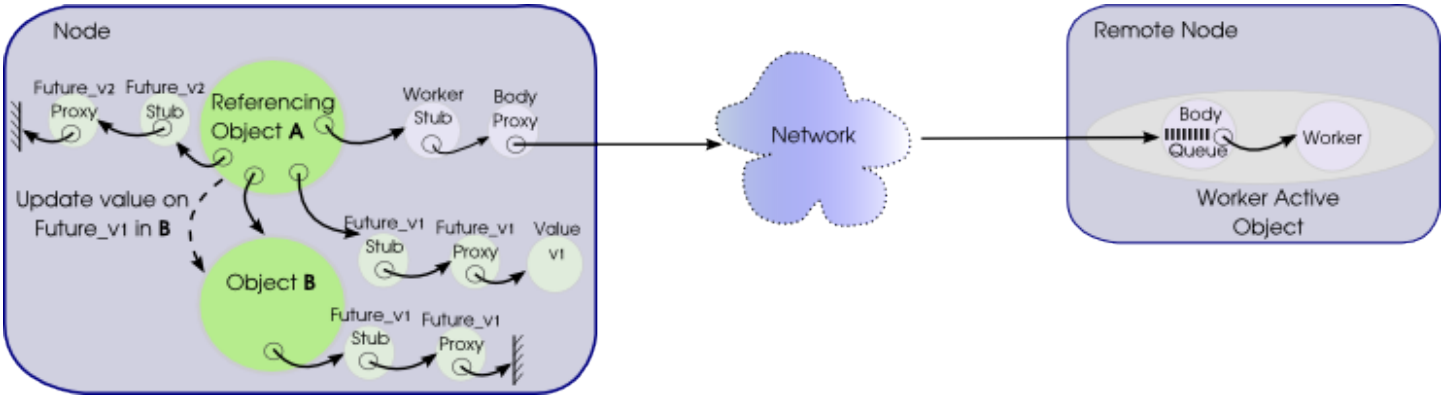
The call on B leads to the creation of another future to which v2 will point to. Since v1 has not been updated yet, there will be another future created on object B.



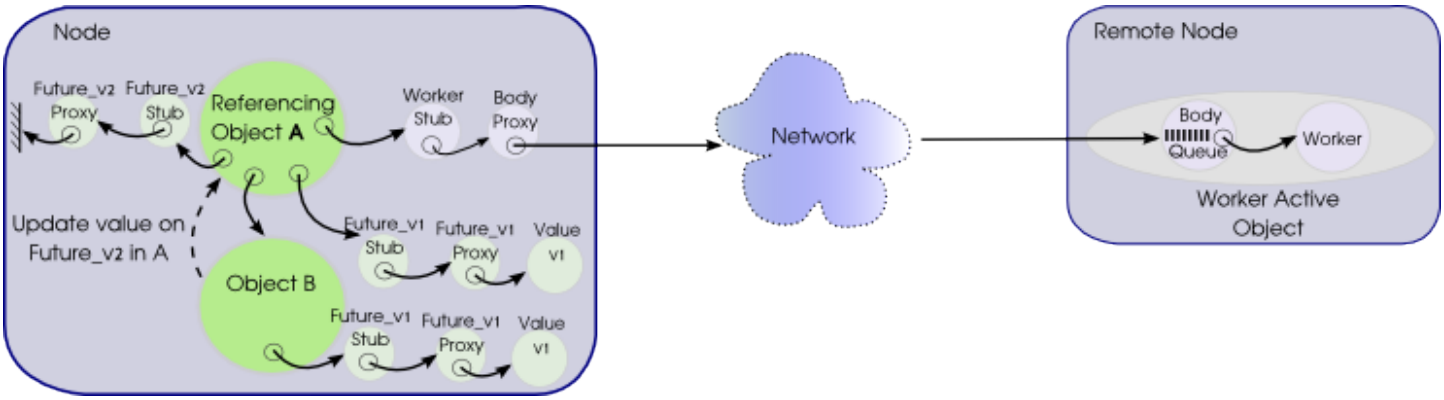
At some point, the active object finishes executing foo() and the value of the future for v1 will be updated by the Body.



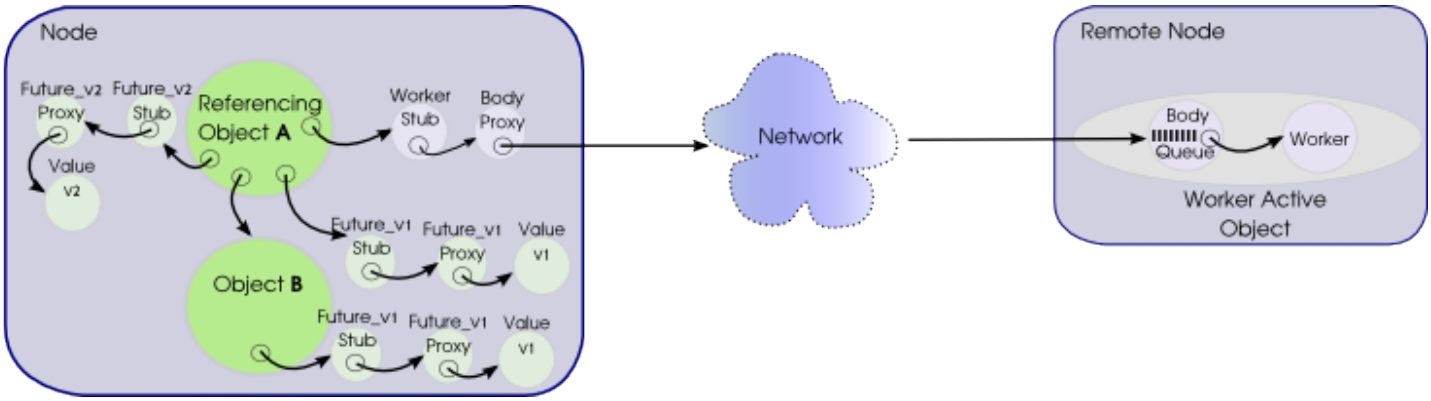
After the value on A is updated, A updates the value in the future on B.



Once B has the value for the future, it will execute bar() with the received value and update the value v2 in A.



The final state is with all the future values updated.



As seen above, as long as there is a future needed for a computation, it will be passed to the object doing the computation when it is not needed immediately. If the value is needed right away, the caller thread will block until the future is updated, just like in single threaded call.

2.9. Data Exchange between Active Objects

Exchange operation offers a way to exchange potentially a very large amount of data between two active objects, making an implicit synchronization. The Exchange operator takes advantage of the serialization / de-serialization steps to exchange data in an efficient way.

Warning

This operator is **NOT** compatible with the ProActive Fault-Tolerance mechanism.

To use it, you have to specify different variables :

- a tag which is an unique identifier for the exchange operation in case of multiple exchanges performed in parallel
- a reference to the active object you want to exchange data with
- a reference to the data source you want to send to the other active object
- a reference to the location where the received data will be put
- the amount of data you want to exchange (must be symmetric)

To illustrate the usage of the Exchange operator, here is an example with two active objects, AO1 and AO2 which are exchanging the half of a local array to get a full one.

```
package org.objectweb.proactive.examples.documentation.activeobjectconcepts;

import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.core.node.NodeException;


public class Exchange {

   public static class AO {
      protected double[] myArray;

      public void display() {
         String vect = "[";
         for (double i : myArray) {
```

```
            vect += i + ", ";
        }
        int index = vect.lastIndexOf(',');
        if (index != -1)
            vect = vect.substring(0, index);

        vect += "]";
        System.out.println(vect);
    }
}

public static class AO1 extends AO {

    public void foo(AO2 ao2) {
        myArray = new double[] { 1, 2, 0, 0 };
        // Before : myArray = [1, 2, 0, 0]
        PAActiveObject.exchange("myExch", ao2, myArray, 0, myArray, 2, 2);
        // After : myArray = [1, 2, 3, 4]
    }
}

public static class AO2 extends AO {

    public void bar(AO1 ao1) {
        myArray = new double[] { 0, 0, 3, 4 };
        // Before : myArray = [0, 0, 3, 4]
        PAActiveObject.exchange("myExch", ao1, myArray, 2, myArray, 0, 2);
        // After : myArray = [1, 2, 3, 4]
    }
}

/**
 * @param args
 */
public static void main(String[] args) {
    try {
        AO1 ao1 = (AO1) PAActiveObject.newActive(AO1.class.getName(), null);
        AO2 ao2 = (AO2) PAActiveObject.newActive(AO2.class.getName(), null);
        ao1.foo(ao2);
        ao2.bar(ao1);
        ao1.display();
        ao2.display();
    } catch (ActiveObjectCreationException e) {
        e.printStackTrace();
    } catch (NodeException e) {
        e.printStackTrace();
    }
}
}
```

Standard exchangeable data are array of integers, doubles and bytes. Also, you can exchange your own complex double structure by implementing the ExchangeableDouble interface. Here is an example of an implementation:

```
package functionalTests.hpc.exchange;
```

```
import org.objectweb.proactive.ext.hpc.exchange.ExchangeableDouble;


/**
 * Implementation example of an {@link ExchangeableDouble} used by the exchange operation.
 */
public class ComplexDoubleArray implements ExchangeableDouble {
    private double[] array;
    private int getPos, putPos;

    public ComplexDoubleArray(int size, boolean odd) {
        this.array = new double[size];
        this.getPos = odd ? 1 : 0;
        this.putPos = odd ? 0 : 1;
        for (int i = getPos; i < array.length; i += 2) {
            array[i] = Math.random();
        }
    }

    public double getChecksum() {
        double sum = 0;
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }

    public String toString() {
        return java.util.Arrays.toString(array);
    }

    public double get() {
        double res = array[getPos];
        getPos += 2;
        return res;
    }

    public boolean hasNextGet() {
        return getPos < array.length;
    }

    public boolean hasNextPut() {
        return putPos < array.length;
    }

    public void put(double value) {
        array[putPos] = value;
        putPos += 2;
    }
}
```

Chapter 3. Typed Group Communication

3.1. 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 such as the use of dedicated interfaces to trigger group communications. We aim at a more flexible mechanism by proposing 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 this way, group communications remains typed.

In order to ease the use of group communications, we provide a set of static methods in the org.objectweb.proactive.api.PAGroup class and a set of methods in the org.objectweb.proactive.core.group.Group interface. ProActive also provides typed group communication, meaning that only methods defined on classes or interfaces implemented by members of the group can be called.

There are two ways to create groups of active objects: using the PAGroup.newGroup(...) static method or using the PAGroup.newGroupInParallel(...) one. Once the group created, it is possible to turn it into active using the PAGroup.turnActiveGroup(...) static method. Group creation takes several parameters similar to those of active object creation. To understand parameters that are not explained here, please refer to Chapter 2, *Active Objects: Creation And Advanced Concepts* as it contains a detailed explanation of every active object creation parameter. To get more information on PAGroup and Group methods, please refer to the JavaDoc[1].

In this chapter, we will describe how to create a group, send a request to members of this group and retrieve the results. For this purpose, we will use the following classes which can be instantiated as an active object:

```
public class A implements Serializable {
  protected String str = "Hello";

  /**
   * Empty no-arg constructor
   */
  public A() {
  }

  /**
   * Constructor which initializes str
   *
   * @param str
   */
  public A(String str) {
    this.str = str;
  }

  /**
   * setStrWrapper
   *
   * @param strWrapper StringWrapper to set
   *      StringWrapper is used since this method gives an example
   *      of parameter scatterring. Parameter need to be serializable.
   * @return this
   */
  public A setStrWrapper(StringWrapper strWrapper) {
```

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../api_published/index.html?index-all.html

```
        this.str = strWrapper.getStringValue();
        return this;
    }

    /**
     * getB
     *
     * @return a B object corresponding to the current Object
     */
    public B getB() {
        if (this instanceof B) {
            return (B) this;
        } else {
            return new B(str);
        }
    }

    /**
     * display str on the standard output
     */
    public void display() {
        System.out.println("A display =====> " + str);
    }

}
```

```
public class B extends A {
    /**
     * Empty no-arg constructor
     */
    public B() {
    }

    /**
     * Constructor which initializes str
     *
     * @param str
     */
    public B(String str) {
        super(str);
    }

    /**
     * display str on the standard output
     */
    public void display() {
        System.out.println("B display =====> " + str);
    }
}
```

3.2. Instantiation Typed Group Creation

Any object which is reifiable can be included in a group. Groups are created using the PAGroup.newGroup(...) static method or the PAGroup.newGroupInParallel(...) one. The common superclass for every group member has to be specified, giving therefore a

minimal type to the group. For instance, using the class A previously defined, the example hereafter shows how to instantiate a group of A's:

```
try {

      A groupA = (A) PAGroup.newGroup(A.class.getName());

} catch (ClassNotReifiableException e) {
   e.printStackTrace();
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

For now, we have just created a group of A's. To know how to fill this group with A object, please refer to Section 3.3, “Group representation and manipulation”.

We can also specify parameters for the creation as shown below. Giving a set of nodes, the group will be created filled with as many A active objects as provided nodes (one A active object per node). In this case, we can also choose parameters for active object creation. Parameters can be either the sames for every A active object or different depending on the node.

The following example points out the two main ways to create a group using a GCM deployment:

```
try {

      /***** GCM Deployment *****/
      File applicationDescriptor = new File(args[0]);
      GCMApplication gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
      gcmad.startDeployment();

      GCMVirtualNode vn = gcmad.getVirtualNode("VN");
      Node node1 = vn.getANode();
      Node node2 = vn.getANode();
      Node[] nodes = new Node[] { node1, node2 };
      /*************************/

      A groupA;
      if (sameParams) {
         /**
          * Same parameters will be used for object creations
          */
         Object[] params = new Object[] { "Node" };
         groupA = (A) PAGroup.newGroup(A.class.getName(), params, nodes);
      } else {
         /**
          * Different parameters will be used for object creations
          */
         Object[][] params = new Object[][] { new Object[] { "Node 1" }, new Object[] { "Node 2" } };
         groupA = (A) PAGroup.newGroup(A.class.getName(), params, nodes);
      }

} catch (ClassNotReifiableException e) {
   e.printStackTrace();
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

Once created, each group member is accessible using the PAGroup.get(Object group, int index) static method:

```
/**
 * Retrieve the two first members of groupA
 */
A firstMember = (A) PAGroup.get(groupA, 0);
A secondMember = (A) PAGroup.get(groupA, 1);
```

It is also possible, once a group instantiated, to turn it into an active object. For that, PAGroup provides several turnActiveGroup(...) methods:

```
/**
 * Turn groupA into an active object
 */
PAGroup.turnActiveGroup(groupA, node1);
```

3.3. Group representation and manipulation

The **typed group representation** we have presented corresponds to the functional view of object groups. 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 has to be used instead. This second representation, **the management representation**, follows a more standard pattern for grouping objects: the Group interface.

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

The following example underlines 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 = PAActiveObject.newActive(A.class, new Object[] { "A Object" });
B b = PAActiveObject.newActive(B.class, new Object[] { "B Object" });

/**
 * For management purposes, get the management representation
 */
Group<A> grp = PAGroup.getGroup(groupA);

/**
 * Add nodes into the group
 * Note: B extends A
 * Note: Objects and active objects are mixed into the group
 */
grp.add(a1);
grp.add(a2);
grp.add(b);

/**
 * Call the display method on every group member
 * Note: only methods of A can be called but we can override
```

```
 * a method of A in B as it is done for the display method.
 */
groupA.display();

/**
 * A new reference to the typed group can also be built as follows
 */
A groupAnew = (A) grp.getGroupByType();
```

It is important to note that elements can be included into a typed group only if their classes are or extend the class specified at the group creation. In our example, an object of class B (B extending A) is included into the group of type A. However, based on Java typing, only the methods defined in the class A can be invoked on the group.

3.4. 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.

```
/**
 * Call the getB method on groupA which returns a group of B
 */
B groupB = groupA.getB();

/**
 * Call the display method on groupB.
 * A wait-by-necessity process is therefore launch in order to
 * wait the end of every getB method previously called on each
 * A member of groupA.
 */
groupB.display();
```

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 PAGroup class enable to execute various forms of synchronisation. For instance: waitOne, waitN, waitAll, waitTheNth, waitAndGetOne. Here is an example:

```
/**
 * Call the getB method on groupA which returns a group of B
 */
B groupB2 = groupA.getB();

/**
 * Wait and capture the first returned member of groupB2
 */
B firstB = (B) PAGroup.waitAndGetOne(groupB2);

/**
 * To wait all the members of vg are arrived
 */
```

```
PAGroup.waitAll(groupB2);
```

3.5. 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 translation 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, the programmer must apply the static method setScatterGroup of the PAGroup class to the group. In order to switch back to the default behaviour, the static method unsetScatterGroup is available.

Here is an example where we define a group of 2 StringWrappers and then call the setStrWrapper method on a group of 3 A's:

```
/**
 * Create a group of parameters
 */
Object[][] params = new Object[][] { new Object[] { "Param 1" }, new Object[] { "Param 2" } };
StringWrapper strWrapper = (StringWrapper) PAGroup.newGroup(StringWrapper.class.getName(),
        params);

/**
 * Call the setScatterGroup method on this group
 */
PAGroup.setScatterGroup(strWrapper);

/**
 * Dispatch parameters
 */
A resultGroup = groupA.setStrWrapper(strWrapper);

/**
 * Call the unsetScatterGroup method on the parameter group
 */
PAGroup.unsetScatterGroup(strWrapper);

/**
 * Call the display method on the result group which displays:
 *
 * A display =====> Param 1
 * A display =====> Param 2
 */
resultGroup.display();

/**
 * Call the display method on the whole group which displays:
 *
 * A display =====> Param 1
 * A display =====> Param 2
```

```
 * B display =====> B Object
 */
groupA.display();
```

We can notice that only the two first elements of groupA have been modified. The last one remains unchanged. If we had defined more parameters than there are group members, a cycle would be operated. In other words, if had defined a parameter group consisted of 4 elements, then the setStrWrapper method would be called twice on the first A member: the first time with the first parameter and the second one with the fourth parameter.

3.6. Access By Name

If also possible, when we insert an element into a group, to specify a name. In that case, this group member will be accessible either by its index as previously or by its name. The example hereafter shows how to insert an element with a name using the addNamedElement(String key,E value) method of Group and how to retrieve this element using its name with the getNamedElement(String) method:

```
A a3 = PAActiveObject.newActive(A.class, new Object[] { "Another A object" });

/**
 * Using the previous management representation, we insert
 * a new A element with the name "nameA"
 */
grp.addNamedElement("namedA", a3);

/**
 * Retrieve this element using its name
 */
A named_a = (A) grp.getNamedElement("namedA");

/**
 * Display it to be sure it's the good one. Returns:
 *
 * A display =====> Another A object
 */
named_a.display();
```

3.7. Unique serialization

Unique serialization is an optimization option that allows performing argument serialization before streaming them. If unique serialization is activated, then arguments will be transformed into a byte array once and streamed afterward. Unique serialization is useful if the size of arguments is significant and considering that bandwidth is more important than the ability to stream serialized objects as soon as possible. This behaviour can be toggled using the setUniqueSerialization(Object) and unsetUniqueSerialization(Object) static methods of PAGroup:

```
/**
 * Set unique serialization
 */
PAGroup.setUniqueSerialization(groupA);

/**
 * Call the display method
 */
groupA.display();
```

```
/**
 * Unset unique serialization
 */
PAGroup.unsetUniqueSerialization(groupA);
```

To learn more about groups, see the JavaDoc[2] of org.objectweb.proactive.core.group and the paper [BBC02].

3.8. Activating a ProActive Group

A group contains the references on its member elements. When a reference on a group is sent from one active object to another one, what is sent is a copy of the group reference. This leads to the existence of different group proxies referencing the same set of elements. These copies are not coherent and when one action is performed on a element of one of the existing instances, the change is not replicated on the others. To overcome this limitation, it is possible to active a group using PAGroup.turnActive(...). The group becomes an active object, a remotely accessible object. The new references are no longer copies of a set of element but a pointer to the active group. The benefits are twofold:

- Consistency: there is only one instance of the group, any change on the group's members is seen by all objects having a reference on it.
- You get a remotely accessible group even if the members are not active objects

Known limitation: An org.objectweb.proactive.core.body.exceptions.InactiveBodyException can be thrown if the following elements are met:

- a class that implements RunActive thus defines runActivity()
- the creation of a group of the previously mentioned class
- the activation of the group using PAGroup.turnActiveAsGroup()

Explanation: It is worth mentioning that a ProActive group is able to perform asynchronous method call. When activated, the group becomes the reified object used within the active object. The initialization logic of an active object calls that runActivity() method if the reified object's class defines it and expects that method to manage the life cycle of the active object (mainly serves incoming requests). When the runActivity() method exits, the body shuts down. When the runActivity() is called on the group, the call is reified into an asynchronous method call while the caller (the active object's thread) continues its execution flow. In other terms, the method runActivity() almost exits immediately from the caller point-of-view leading the active object in the shutdown sequence.

To overcome this limitation, the solution is to create an intermediary class in the inheritance graph that does not implement RunActive and use this class as base class for the group.

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../api_published/index.html

Chapter 4. Mobile Agents And Migration

4.1. Migration Overview

Active objects have the ability to migrate between nodes while the application is running. Whatever application is used the active objects will continue running regardless of the migration of active objects.

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

In order to ease the use of the migration, ProActive provides 2 sets of static methods in the PAMobileAgent 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:

- PAMobileAgent.migrateTo(Object o): migrate to the same location as an existing active object
- PAMobileAgent.migrateTo(String nodeURL): migrate to the location given by the URL of the node
- PAMobileAgent.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. The boolean value in the following methods indicates whether the migration modifies the computation of the application.

- PAMobileAgent.migrateTo(Body body, Object o, boolean isNFRequest): migrate to the same location as an existing active object
- PAMobileAgent.migrateTo(Body body, String nodeURL, boolean isNFRequest): migrate to the location given by the URL of the node
- PAMobileAgent.migrateTo(Body body, Node node, boolean isNFRequest): migrate to the location of the given node
- PAMobileAgent.migrateTo(Body body, Node node, boolean isNFRequest, int priority): migrate to the location of the given node The int value indicates the priority of the non functional request. Levels are defined in the Request interface.

4.2. 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 has to implement the Serializable interface**. To migrate, an active object has to implement a method which contains a call to PAMobileAgent.migrateTo(...). This call must be the last one in the method, i.e the method has to return immediately after. Here is an example of a method in an active object:

```
/**
 * Migrates the active object
 *
 * @param url - URL of the node where you want to
 *              migrate your active object to
 */
public void moveTo(String url) {
    try {
        PAMobileAgent.migrateTo(url);
    } 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. To make this object migrate, you just have to call its moveTo(String) method.

4.3. Complete example

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
import java.net.InetAddress;

import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.api.PAMobileAgent;


/**
 * @author The ProActive Team
 *
 */
public class SimpleAgent implements Serializable {

   /**
    * Empty, no-arg constructor required by ProActive
    */
   public SimpleAgent() {
   }

   /**
    * Migrates the active object
    *
    * @param url - URL of the node where you want to
    *            migrate your active object to
    */
   public void moveTo(String url) {
      try {
         PAMobileAgent.migrateTo(url);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }


   /**
    * Retrieves the name of the host where the active
    * object is.
    *
    * @return The node url and the hostname where the active object is
    */
   public String whereAreYou() {
      try {
         return PAActiveObject.getActiveObjectNodeUrl(PAActiveObject.getStubOnThis()) + " on host " +
            InetAddress.getLocalHost().getHostName();
      } catch (Exception e) {
```

```
            return "Localhost lookup failed";
        }
    }
}
```

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.

```
public static void simpleMigration(String[] args) {
    if (args.length != 1) {
        System.out
                .println("Usage: java org.objectweb.proactive.examples.documentation.migration.Migration
 GCMA.xml");
        System.exit(-1);
    }

    // Gets a node from a deployment and
    // creates the SimpleAgent on this node

    /***** GCM Deployment *****/
    File applicationDescriptor = new File(args[0]);

    GCMApplication gcmad;
    try {
        gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
    } catch (ProActiveException e) {
        e.printStackTrace();
        return;
    }
    gcmad.startDeployment();

    // Take a node from the available nodes of VN
    GCMVirtualNode vn = gcmad.getVirtualNode("VN");
    vn.waitReady();
    Node node = vn.getANode();
    /*************************/

    SimpleAgent simpleAgent;
    try {
        // Creates the SimpleAgent in this JVM
        simpleAgent = PAActiveObject.newActive(SimpleAgent.class, null, node);
    } catch (Exception e) {
        e.printStackTrace();
        return;
    }

    System.out.println("The Active Object has been created on " + simpleAgent.whereAreYou());

    // Migrates the SimpleAgent to another node
    node = vn.getANode();
    simpleAgent.moveTo(node.getNodeInformation().getURL());
    System.out.println("The Active Object is now on " + simpleAgent.whereAreYou());
}
```

In the main method, we first need to create an active object, which is done through the call to newActive(). Once done, we load an application description in order to get a node. Then, we call its moveTo method which will make it migrate to the node specified as parameter and finally, we ask it what is its current location. For our application, we pass the location of the node as runtime parameter to the application.

To see another example of migration look at the migration example in Chapter 6.5. Monitoring Several Computers Using Migration[1].

4.4. Dealing with non-serializable attributes

The migration of an active object uses serialization. Unfortunately, not all the objects in the Java language are serializable so the first step to solve the problem is to identify the unserializable components. In front of the declaration of unserializable fields, we will put the keyword transient. This indicates that the value of this variable should not be serialized. After the first migration, these fields will be set to null since they have not been saved. So we have to explicitly rebuild them upon arrival of the active object at its new location. This can be done by using the MigrationStrategyManager interface or by implementing readObject(java.io.ObjectInputStream in) and writeObject(java.io.ObjectOutputStream out) from the interface Serializable.

4.4.1. Using The MigrationStrategyManager Interface

We can use the methods onArrival(String) and onDeparture(String) to specify which methods will be run when the active objects migrates.

The arguments for onArrival(String) and onDeparture(String) are respectively the methods to be run before serialization and after deserialization. We instantiate a new MigrationStrategyManagerImpl for the body of the object to be migrated and assign the methods to be run. For this, we use the initActivity(Body) method since we only need to run the creation and assignment once.

Here is the code of our UnSerializableAgent class:

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
import java.math.BigInteger;
import java.net.InetAddress;
import java.util.Random;

import org.objectweb.proactive.Body;
import org.objectweb.proactive.InitActive;
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.api.PAMobileAgent;
import org.objectweb.proactive.core.migration.MigrationStrategyManagerImpl;


/**
 * @author The ProActive Team
 *
 */
public class UnSerializableAgent implements InitActive, Serializable {

   private byte[] saved; //will store the values
   private transient BigInteger toBeSaved; //will be null after serialization
```

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../GetStarted/multiple_html/ActiveObjectTutorial.html#MonitoringSeveralComputersUsingMigration_18

```
/**
 * Empty, no-arg constructor required by ProActive
 */
public UnSerializableAgent() {
}

public UnSerializableAgent(int length) {
    toBeSaved = new BigInteger(length, new Random());
}

public void displayArray() {
    System.out.println(toBeSaved.toString());
}

/**
 * Migrates the active object
 *
 * @param url - URL of the node where you want to
 *              migrate your active object to
 */
public void moveTo(String url) {
    try {
        PAMobileAgent.migrateTo(url);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * Retrieves the name of the host where the active
 * object is.
 *
 * @return The node url and the hostname where the active object is
 */
public String whereAreYou() {
    try {
        return PAActiveObject.getActiveObjectNodeUrl(PAActiveObject.getStubOnThis()) + " on host " +
            InetAddress.getLocalHost().getHostName();
    } catch (Exception e) {
        return "Localhost lookup failed";
    }
}

public void initActivity(Body body) {
    MigrationStrategyManagerImpl myStrategyManager = new MigrationStrategyManagerImpl(
        (org.objectweb.proactive.core.body.migration.Migratable) body);
    myStrategyManager.onArrival("rebuild");
    myStrategyManager.onDeparture("leaveHost");
}

/**
 * Restore values of unserializable variable
 * that have been previously saved
```

```
     */
    public void rebuild() {
        toBeSaved = new BigInteger(saved);
        System.out.println("Unserializable variables has been restored");
    }

    /**
     * Saves the values of the unserializable variables
     */
    public void leaveHost() {
        saved = toBeSaved.toByteArray();
        System.out.println("Unserializable variables has been saved");
    }
}
```

This class is normally non-serializable since BigInteger is not serializable. If we tried to migrate the active object without implementing the initActivity() method as previously, we would get a NullPointerException when invoking a method on our toBeSaved variable.

The following method is in charge of instantiate our object and to migrate it:

```
public static void unserializableMigration(String[] args) {
    if (args.length != 1) {
        System.out
                .println("Usage: java org.objectweb.proactive.examples.documentation.migration.Migration
 GCMA.xml");
        System.exit(-1);
    }

    // Gets a node from a deployment and
    // creates the SimpleAgent on this node

    /***** GCM Deployment *****/
    File applicationDescriptor = new File(args[0]);

    GCMApplication gcmad;
    try {
        gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
    } catch (ProActiveException e) {
        e.printStackTrace();
        return;
    }
    gcmad.startDeployment();

    // Take a node from the available nodes of VN
    GCMVirtualNode vn = gcmad.getVirtualNode("VN");
    vn.waitReady();
    Node node = vn.getANode();
    /*************************/

    UnSerializableAgent unserializableAgent;
    try {
        // Creates the SimpleAgent in this JVM
        Object[] params = new Object[] { 100 };
        unserializableAgent = PAActiveObject.newActive(UnSerializableAgent.class, params, node);
```

```
    } catch (Exception e) {
        e.printStackTrace();
        return;
    }

    System.out.println("The Active Object has been created on " + unserializableAgent.whereAreYou());
    unserializableAgent.displayArray();

    // Migrates the SimpleAgent to another node
    node = vn.getANode();
    unserializableAgent.moveTo(node.getNodeInformation().getURL());
    System.out.println("The Active Object is now on " + unserializableAgent.whereAreYou());
    unserializableAgent.displayArray();
}
```

4.4.2. Using readObject(...) and writeObject(...)

Another way to deal with unserializable attributes is using the methods from the Serializable interface:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;
private void readObjectNoData() throws ObjectStreamException;
```

We override these methods to provide a way to save and restore the non-serializable components of the active object:

```
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;
import java.math.BigInteger;
import java.net.InetAddress;
import java.util.Random;

import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.api.PAMobileAgent;


/**
 * @author The ProActive Team
 *
 */
public class UnSerializableAgent2 implements Serializable {

    private transient BigInteger toBeSaved; //will be null after serialization

    /**
     * Empty, no-arg constructor required by ProActive
     */
    public UnSerializableAgent2() {
    }

    public UnSerializableAgent2(int length) {
        toBeSaved = new BigInteger(length, new Random());
```

```
    }

    public void displayArray() {
        System.out.println(toBeSaved.toString());
    }

    /**
     * Migrates the active object
     *
     * @param url - URL of the node where you want to
     *              migrate your active object to
     */
    public void moveTo(String url) {
        try {
            PAMobileAgent.migrateTo(url);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * Retrieves the name of the host where the active
     * object is.
     *
     * @return The node url and the hostname where the active object is
     */
    public String whereAreYou() {
        try {
            return PAActiveObject.getActiveObjectNodeUrl(PAActiveObject.getStubOnThis()) + " on host " +
                InetAddress.getLocalHost().getHostName();
        } catch (Exception e) {
            return "Localhost lookup failed";
        }
    }

    private void writeObject(java.io.ObjectOutputStream out) throws java.io.IOException {
        //do the default serialization
        out.defaultWriteObject();
        //do the custom serialization
        out.writeObject(toBeSaved);
    }

    private void readObject(java.io.ObjectInputStream in) throws java.io.IOException, ClassNotFoundException {
        //do the default deserialization
        in.defaultReadObject();
        //do the custom deserialization
        toBeSaved = (BigInteger) in.readObject();
    }
}
```

To learn how to use the methods, see the Serializable[2] interface in the standard JavaDoc.

[2] http://java.sun.com/javase/6/docs/api/java/io/Serializable.html

4.5. Mixed Location Migration

4.5.1. Forwarders And Agents

There are two ways to communicate with an active object which has migrated:

- **Forwarders**

  An active object, which is about to leave a site, leaves behind a special object called a forwarder which is in charge of forwarding incoming messages to the next destination. As time goes, a chain of forwarders is built between a caller and the mobile object. Any message sent to the latter will go through this chain to reach the agent. There is a virtual path between a caller and a mobile object. As soon as a call is performed on the agent, the chain of forwarders is shortened into a direct link to the agent called.

- **Location Server**

  Communicating with a mobile object can be done with an explicit reference towards the mobile entity, which requires the means to get its current location when necessary.

  There is a two-step communication: first there should be a search to obtain an up-to-date reference (localization), and then the actual communication. The simplest solution is to have a unique location server which maintains a database of the known position of the agents. When an object wants to communicate with an object which has migrated, it queries the server which sends back a new reference. If this is a correct reference then the communication takes place, otherwise a new query is issued.

Both techniques have their drawbacks. Two problems arise when using a forwarding scheme, especially if the ambition is scalable mobile agents over WAN. First, the forwarders use resources on a site as long as they have not been garbage collected. Thus, if a chain exists between to objects, it will remains even if there is no new communications going by. Second, the longer the chain is, the more likely it will be cut because of a hardware or software failure. As a consequence, while forwarders are more efficient under some conditions, they do not appear to be scalable, nor reliable.

On the other hand, the server is a single point of failure and a potential bottleneck. If a server is to help communicating with a higher number of mobile agents, then it might not be able to serve requests quickly enough. Furthermore, in case of a crash, it is not possible to communicate with mobile active objects until the server is back. It is possible to avoid most of these issues by having redundant servers with load balancing at the cost of increasing complexity.

Based on these observations and taking into account the variability of the environment, we use a configurable communication protocol which offers the main benefits from both the forwarder and the server while avoiding their drawbacks. Configurable with discrete and continuous parameters, it can be tailored to the environment to offer both performance and reliability.

4.5.2. Forwarder And Agent Parameters

Forwarders and agents have several parameters which control their behaviour:

- **forwarder time to live in milliseconds** - proactive.mixedlocation.ttl

  Time limited forwarders remain alive only for a limited period. When their lifetime is over, they can be removed. First of all, this brings an important advantage: scalability due to absence of the DGC and the systematic reclaim of forwarding resources. However this increases the risks of having the forwarding chain cut since a forwarder might expire during the normal execution of the application. In such a situation, we rely on a server which will be considered as an alternative solution. This increases the overall reliability.

  Use -1 to indicate that the forwarders have a unlimited lifetime.

- **forwarder updating status, boolean** - proactive.mixedlocation.updatingForwarder

  It is possible to rely on the forwarders to maintain the location of the agent by having them update the server. When they reach the end of their lifetime, they can send to the server their outgoing reference which could be the address of the agent or another forwarder. The updatingForwarder parameter can be true or false. If true it releases the agent from most of the updates. In order to increase reliability, it is possible to have also the agent update the server on a regular basis.

- **agent time to update in milliseconds**

  Each mobile agent has a nominal Time To Update (TTU) after which it will inform the localization server of its new location. The TTU is defined as the first occurrence of two potential events since the last update:

  - **agent migrations number, integer** - proactive.mixedlocation.maxMigrationNb

    This parameter indicates the number of migrations before updating the server. Use -1 to indicate that the agent never updates the server.

  - **agent time on site in milliseconds** - proactive.mixedlocation.maxTimeOnSite

    The parameter indicates the maximum time spent on a site before updating the server. You can use -1 to indicate that agent will not inform the server.

If we consider that both the agent and the forwarders can send updates to the server, the server must be able to make the difference between messages from the forwarders and from the agent which are always the most up to date. Also, since we do not have any constraint on the Time To Live (TTL) of the forwarders, it could be that a forwarder at the beginning of a chain dies after on at the end. If this happens and we are not careful when dealing with the requests, the server could erase a more up to date reference with an old one.

4.5.3. Configuration File

As a default, ProActive uses a "Forwarder based" strategy. It means that the forwarders have a unlimited lifetime and the agent never updates the location server.

To configure your own strategy, you have to edit the file ProActive/src/Core/org/objectweb/proactive/core/config/ProActiveConfiguration.xml and change the four values listed above.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveUserProperties>
  <properties>
    <prop key="proactive.locationserver"
      value="org.objectweb.proactive.ext.locationserver.LocationServer" />
    <prop key="proactive.locationserver.rmi" value="//localhost/LocationServer" />
    <prop key="proactive.securitymanager" value="true" />
    <prop key="proactive.mixedlocation.ttl" value="-1" />
    <prop key="proactive.mixedlocation.updatingForwarder"
      value="false" />
    <prop key="proactive.mixedlocation.maxMigrationNb" value="-1" />
    <prop key="proactive.mixedlocation.maxTimeOnSite" value="-1" />
     <prop key="proactive.implicitgetstubonthis" value="false"/>

    <prop key="proactive.net.disableIPv6" value="true" />

    <prop key="proactive.future.ac" value="true" />
    <prop key="proactive.stack_trace" value="false" /> <!-- complete stack traces in requests -->
    <prop key="proactive.dgc" value="false" />
    <prop key="proactive.exit_on_empty" value="false" /> <!-- destroy the JVM when there is no more active object
-->
    <prop key="schema.validation" value="true" />
    <prop key="proactive.communication.protocol" value="rmi" />
    <prop key="proactive.rmi.port" value="1099" />

   <!--<prop key="proactive.java.policy" value="../../../../../../scripts/unix/proactive.java.policy"/>-->
    <prop key="gcm.provider" value="org.objectweb.proactive.core.component.Fractive" />
   <!-- <prop key="proactive.tunneling.connect_timeout" value="2000"/> -->
    <prop key="proactive.communication.rmissh.try_normal_first" value="false" />
```

```
    <!-- <prop key="proactive.communication.benchmark.class"
value="org.objectweb.proactive.core.remoteobject.benchmark.LargeByteArrayBenchmark"/> -->
    <prop key="proactive.classloading.useHTTP" value="true"/>

    <!-- SSL cipher suites used for RMISSL communications.
  List of cipher suites used for RMISSL, separated by commas.
  default is SSL_DH_anon_WITH_RC4_128_MD5. This cipher suite is used only
  to have encrypted communications, without authentication, and works with default
  JVM's keyStore/TrustStore

  Many others cipher suites can be used. for implementing a certificate authentication...
  see http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html -->
    <prop key="proactive.ssl.cipher.suites" value="SSL_DH_anon_WITH_RC4_128_MD5"/>


    <!-- <prop key="proactive.ssh.known_hosts" value="/user/mlacage/home/.ssh/known_hosts"/> -->
    <!-- <prop key="proactive.ssh.key_directory" value="/user/mlacage/home/.ssh/"/> -->
    <!-- <prop key="proactive.ssh.port" value="22"/> -->
    <!-- <prop key="proactive.ssh.username" value="mlacage"/> -->
    <prop key="components.creation.timeout" value="10000" /> <!-- timeout in seconds for parallel creation of
components -->
    <!-- ************************************************** -->
    <!-- MASTER/WORKER PROPERTIES VALUE -->
    <!-- Period of the slave pinging (in ms) -->
    <prop key="proactive.masterworker.pingperiod" value="10000" />
    <!-- Compressing tasks in the repository -->
    <prop key="proactive.masterworker.compresstasks" value="false" />

    <!-- Message Tagging -->
    <prop key="proactive.tagmemory.lease.max" value="60" />
    <prop key="proactive.tagmemory.lease.period" value="21" />
    <prop key="proactive.tag.dsf" value="false" />

    <!-- Workaround to fix rmissh conflict -->
    <prop key="proactive.runtime.stayalive" value="true" />

    <!-- File Transfer Properties -->
    <prop key="proactive.filetransfer.services_number" value="16" />
    <prop key="proactive.filetransfer.blocks_number" value="8" />
    <prop key="proactive.filetransfer.blocks_size_kb" value="512" />
    <prop key="proactive.filetransfer.buffer_size_kb" value="256" />

    <prop key="proactive.test.perf.duration" value="30000" />


<prop key="proactive.log4j.appender.provider" value="org.objectweb.proactive.core.util.log.remote.ThrottlingProvider"
/>

    <prop key="proactive.gcmd.unix.shell" value="/bin/sh"/>

    <prop key="proactive.mop.writestubondisk" value="false"/>

    <prop key="proactive.webservices.framework" value="cxf" />
    <prop key="proactive.webservices.elementformdefault" value="false" />
```

```
  </properties>
  <javaProperties>
    <!-- <prop key="java.rmi.server.RMIClassLoaderSpi"
 value="org.objectweb.proactive.core.classloading.protocols.ProActiveRMIClassLoader"/> -->
    <prop key="java.protocol.handler.pkgs" value="org.objectweb.proactive.core.ssh|
org.objectweb.proactive.core.classloading.protocols" />
    <prop key="ibis.name_server.host" value="localhost" />
    <prop key="ibis.name_server.key" value="1" />
    <prop key="ibis.io.serialization.classloader" value="org.objectweb.proactive.core.mop.MOPClassLoader" />
    <prop key="ibis.serialization" value="ibis" />
   </javaProperties>
</ProActiveUserProperties>
```

4.5.4. Location Server

A location server is available in the package org.objectweb.proactive.core.body.migration.MixedLocationServer. A simpler one is also available in the package org.objectweb.proactive.ext.util.SimpleLocationServer. The use of location is a little more cumbersome. If want to use this functionality, please refer to the functionalTests.activeobject.locationserver.TestLocationServer functional test. It gives an implementation of an active object migration using the SimpleLocationServer class as location server.

Limitation of this functionality: there can be only one LocationServer for the migration.

Chapter 5. Exception Handling

5.1. Exceptions and Asynchrony

In the asynchronous environment provided by ProActive, exceptions cannot be handled in the same manner as in a sequential environment. Let's see the problem with exceptions and asynchrony in a piece of code. The following method is a simple method on class A that throws an exception according to the boolean parameter.

```
/**
 * Example of a method which can throw an exception
 *
 * @param hasToThrow boolean saying whether the method should
 *                   throw an exception
 * @throws Exception
 */
public void throwsException(boolean hasToThrow) throws Exception {
    Thread.sleep(5000);
    if (hasToThrow)
        throw new Exception("Class A has thrown an exception");
}
```

As you can see, we have inserted a waiting time. This is done in order to examine the asynchronous/synchronous behaviour. With a common exception handling, we would proceed as follows:

```
System.out.println("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv");
try {
    A a = PAActiveObject.newActive(A.class, null);
    a.throwsException(true); //Synchronous method due to the potential exception
    System.out.println("Hello");
    //...
} catch (Exception e) {
    e.printStackTrace();
}
System.out.println("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
```

As the method call in line 2 is asynchronous, the program continues its execution without waiting for its completion. So, it is possible for the control flow to exit the try. In that case, if the method call finishes with an exception, we cannot throw it back in the code anymore because we are no more in the try block. That is why ProActive method calls with potential exceptions are handled synchronously by default. If you try this example, you will not see the Hello display which shows that the method was synchronous and thus, the exception has been catch immediately.

5.1.1. 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 thrown at several points:

- when accessing a future
- inside the barrier
- by using the provided API (see after)

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

```
System.out.println("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv");
PAException.tryWithCatch(Exception.class);
try {
   A a = PAActiveObject.newActive(A.class, null);
   a.throwsException(true); // Asynchronous method call that can throw an exception
   System.out.println("Hello");
   //...
   PAException.endTryWithCatch();
} catch (Exception e) {
   e.printStackTrace();
} finally {
   PAException.removeTryWithCatch();
}
System.out.println("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
```

With this code, the method call 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. If you try this piece of code, you will see that, contrary to the previous example, Hello will appear and then you will see the caught exception.

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

- PAException.tryWithCatch() - right before the try block. The parameter is either the caught exception class or an array of exception classes if there are several one.
- PAException.endTryWithCatch() - at the end of the try block.
- PAException.removeTryWithCatch() - at the beginning of the finally block, so the block becomes mandatory.

5.1.2. 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 into the second one. Here is a sample session with the tool:

```
$ ProActive/bin/trywithcatch.sh 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) {
+    PAException.tryWithCatch(AnException.class);
       try {
         return a.aMethod();
+         PAException.endTryWithCatch();
       } catch (AnException ae) {
         return null;
+      } finally {
+         PAException.removeTryWithCatch();
       }
   }
 }
```

As we can see, ProActive method calls are added to make sure all calls within try/catch blocks are handled asynchronously. The tool can be found in ProActive/bin/ and is named trywithcatch.[sh|bat]. The default behaviour of trywithcatch.[sh|bat] is to add only PAException... without importing the PAException's package. If you do not want to bother with this manual

importation, launch this script with the -fullname option. That will write the full name of PAException, that is to say, org.objectweb.proactive.api.PAException.

5.1.3. throwArrivedException() and waitForPotentialException()

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

The first method is PAException.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 PAException.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.

```
System.out.println("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv");
PAException.tryWithCatch(Exception.class);
try {
   A a = PAActiveObject.newActive(A.class, null);
   a.throwsException(true); // Asynchronous method call that can throw an exception
   //...
   // Throws exceptions which has been already raised by active object
   PAException.throwArrivedException();

   // Should appear since the exception did not have the time
   // to be thrown (due to the sleep(5000))
   System.out.println("Hello");
   //...
   try {
      Thread.sleep(5000);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
   // Throws exceptions which has been already raised by active object
   PAException.throwArrivedException();

   // Should not appear since the exception had the time to
   // be thrown.
   System.out.println("Hello");
   System.out.println("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
   PAException.endTryWithCatch();
} catch (Exception e) {
   e.printStackTrace();
} finally {
   PAException.removeTryWithCatch();
}
```

If you run the previous, you will see that the method behaviour is dependent on the timing since the first call does not throw exception whereas the second one throws one. Thus, calling this method may or may not result in an exception being thrown, depending on the time for an exception to come back. That is why another method is provided: PAException.waitForPotentialException(). Unlike the previous PAException.throwArrivedException(), 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.

```
System.out.println("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv");
PAException.tryWithCatch(Exception.class);
try {
   A a = PAActiveObject.newActive(A.class, null);
```

```
    a.throwsException(true); // Asynchronous method call that can throw an exception
    //...
    // At that moment, we want to be sure that no exception has been
    // raised by the code before.
    PAException.waitForPotentialException();
    //...
    // Should not appear since waitForPotentialException
    // is blocking.
    System.out.println("Hello");
    System.out.println("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
    PAException.endTryWithCatch();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    PAException.removeTryWithCatch();
}
```

Part II. Programming With High Level APIs

Table of Contents

Chapter 6. Master-Worker API

6.1. Overview

Master-Worker computations are the most common case of distributed computations. They are suited well for embarrassingly parallel problems, in which the problem is easy to segment into a very large number of parallel tasks, and no essential dependency (or communication) between those parallel tasks are required.

The main goal of the ProActive Master-Worker API is to provide an easy to use framework for parallelizing embarrassingly parallel applications.

The main features are:

- Automatic tasks scheduling for the Workers.
- Automatic load-balancing between the Workers
- Automatic fault-tolerance mechanism (i.e. when a Worker is missing, the task is rescheduled)
- Very simple mechanism for solution gathering
- All the internal concepts of ProActive are hidden from the user
- Open and extensible API

The usage of the Master-Worker API is simple and consists of four steps:

1. Deployment of the Master-Worker framework.
2. Task definition and submission
3. Results gathering
4. Release of acquired resources

6.2. Logger Settings

Before using the Master-Worker, launching examples or writing your own code, it is very useful to enable the maximum logging information to have a deeper look at how the API works. In order to do that you will need to add the following lines in the proactive-log4j file you are using:

```
log4j.logger.proactive.masterworker = DEBUG
log4j.additivity.proactive.masterworker = false
```

6.3. Master Creation And Deployment

When creating the master, the user application has the possibility to create either a local master (on the machine the user application is running on) or a remote one. Regardless of the way it is created, the active object instantiation is transparent. The deployment process can be controlled by using the deployment descriptors.

The deployment of the Master-Worker framework relies on the ProActive deployment mechanism. In order to deploy a set of workers, the master needs either:

- a ProActive deployment descriptor to be used by the master to deploy its resources
- a set of already deployed ProActive resources like a VirtualNode object or a Collection of Node objects

For a full explanation of the ProActive deployment mechanism and of ProActive deployment descriptors, please refer to Chapter 15, *XML Deployment Descriptors*.

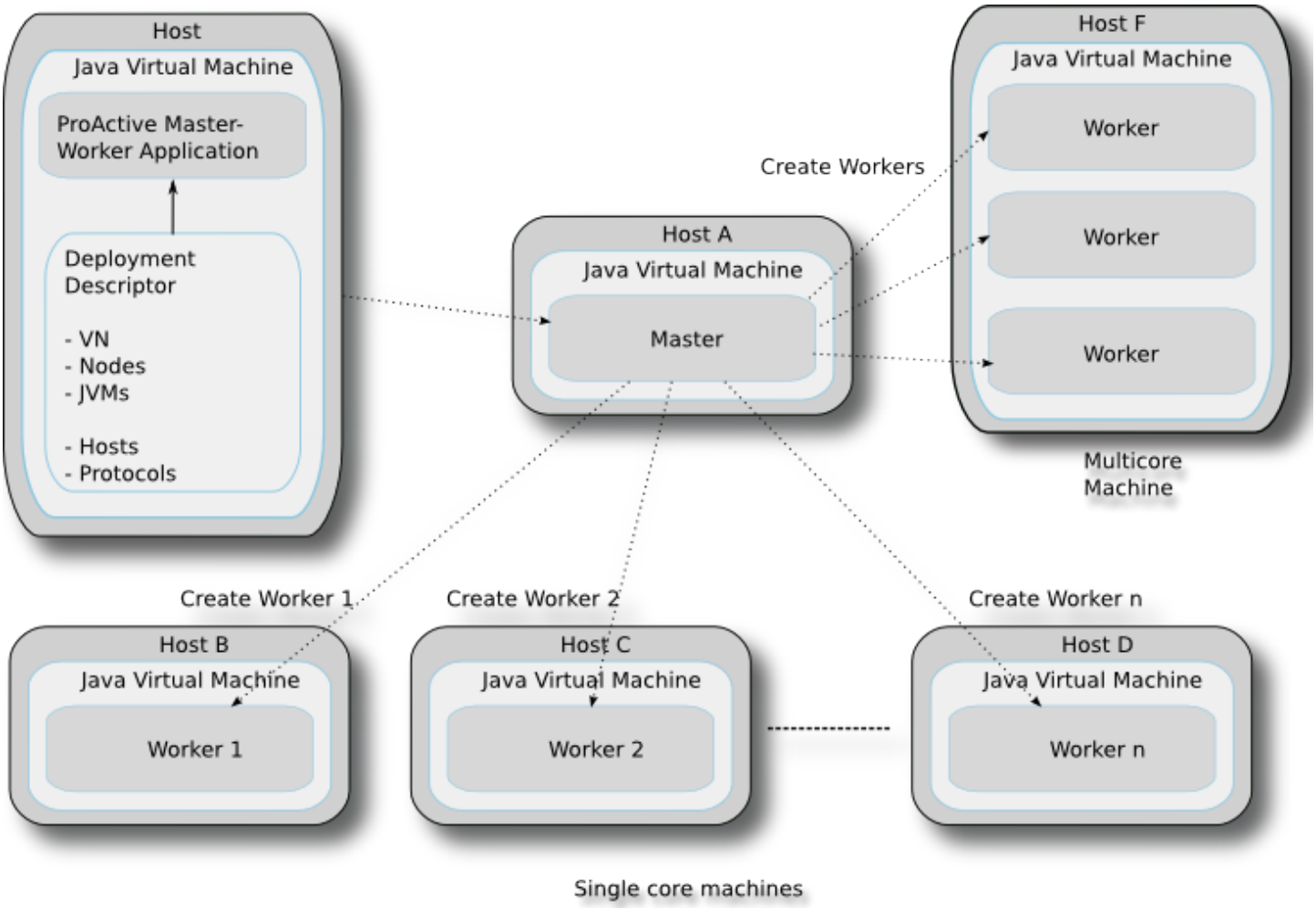


Figure 6.1. Deployment of the Master-Worker framework

The figure represents one case of deployment as it is possible to have several workers on a single core machine each running in its own independent thread, one worker in its own JVM on a multi-core machine, or workers deployed on the machine running the master. The worker-JVM-machine mappings depend on the deployment descriptor. The only restriction is that each Worker is always started in its own Node. However, from the user application perspective, this is transparent as the Master performs communication and loads balancing automatically. In the following sections, we will consider only the case of several single core machines since the same principles apply regardless of machine type.

6.3.1. Local Master creation

In order to create a local master, we use a constructor without parameters:

```
master = new ProActiveMaster<A, Integer>();
```

Using this constructor, a master will be created in the current JVM and it will share CPU usage and memory with the user JVM. This master will compute tasks of type A and will produce Integer objects as a results.

6.3.2. Remote Master creation

In order to create a remote master the following constructors can be used:

```
/**
 * Creates a remote master that will be created on top of the given Node <br>
 * Resources can be added to the master afterwards
 *
 * @param remoteNodeToUse this Node will be used to create the remote master
 */
public ProActiveMaster(Node remoteNodeToUse)
```

```
/**
 * Creates an empty remote master that will be created on top of the given Node with an initial worker memory
 *
 * @param remoteNodeToUse this Node will be used to create the remote master
 * @param memoryFactory factory which will create memory for each new workers
 */
public ProActiveMaster(Node remoteNodeToUse, MemoryFactory memoryFactory)
```

Using either of these constructors, a master will be created in the specified remote resource(JVM), the master will share CPU usage and memory with existing running applications on the remote host. The mechanism in use to deploy the master remotely is the ProActive deployment mechanism (see Chapter 15, *XML Deployment Descriptors* for further details).

6.4. Adding Resources

Now that the master has been created, resources (Workers) must be added to it. The following methods can be used for creating workers:

```
/**
 * Adds the given Collection of nodes to the master <br/>
 * @param nodes a collection of nodes
 */
void addResources(Collection<Node> nodes);

/**
 * Adds the given descriptor to the master<br>
 * Every virtual nodes inside the given descriptor will be activated<br/>
 * @param descriptorURL URL of a deployment descriptor
 * @throws ProActiveException if a problem occurs while adding resources
 */
void addResources(URL descriptorURL) throws ProActiveException;

/**
* Adds the given descriptor to the master<br>
* Every virtual nodes inside the given descriptor will be activated<br/>
* @param descriptorURL URL of a deployment descriptor
* @param contract a variable contract for this descriptor
* @throws ProActiveException if a problem occurs while adding resources
*/
void addResources(URL descriptorURL, VariableContract contract) throws ProActiveException;

/**
 * Adds the given descriptor to the master<br>
 * Only the specified virtual node inside the given descriptor will be activated <br/>
 * @param descriptorURL URL of a deployment descriptor
 * @param contract a variable contract for this descriptor
 * @param virtualNodeName name of the virtual node to activate
 * @throws ProActiveException if a problem occurs while adding resources
```

```
 */
void addResources(URL descriptorURL, VariableContract contract, String virtualNodeName)
        throws ProActiveException;

/**
* Adds the given descriptor to the master<br>
* Only the specified virtual node inside the given descriptor will be activated <br/>
* @param descriptorURL URL of a deployment descriptor
* @param virtualNodeName name of the virtual node to activate
* @throws ProActiveException if a problem occurs while adding resources
*/
void addResources(URL descriptorURL, String virtualNodeName) throws ProActiveException;

/**
* Connects this master to a ProActive Scheduler<br>
* <br/>
* @param schedulerURL URL of a scheduler
* @param login scheduler username
* @param password scheduler password
* @param classpath an array of directory or jars that will be used by the scheduler to find user classes (i.e. tasks
 definitions)
* @throws ProActiveException if a problem occurs while adding resources
*/
void addResources(String schedulerURL, String login, String password, String[] classpath)
        throws ProActiveException;
```

The first two methods will tell the master to create workers on already deployed ProActive resources. The last two methods will ask the master to deploy resources using a ProActive descriptor and to create workers on top of these resources. For a complete explanation of ProActive's deployment mechanism, please refer to Chapter 15, *XML Deployment Descriptors* .

6.5. Tasks definition and submission

Tasks are submitted through classes that implement the Task interface. In this interface, the unique method run will contain the code to be executed remotely. After the tasks have been submitted to the master, the master will dispatch them automatically to the workers.

Warning

When a Java object implementing the Task interface (i.e. a user task) is submitted to the master, the object will be deep-copied to the master. In consequence, every referenced objects will also be copied. When tasks are submitted to the remote workers, the user task objects will be **serialized** and sent through the network. As a consequence, information which have only local meaning will be lost (database connections, references to etc.)

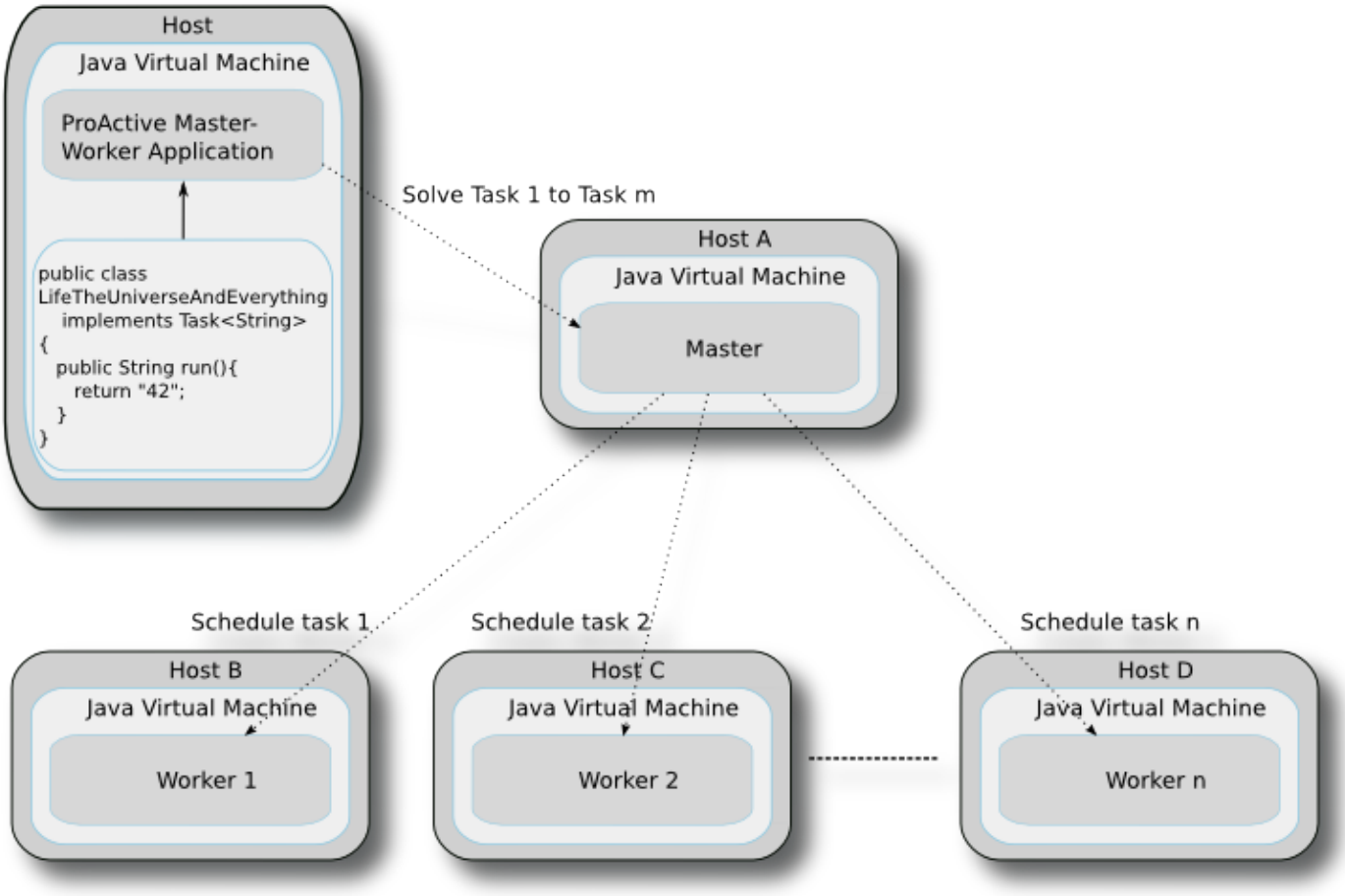


Figure 6.2. Tasks definition and submission

The task interface is org.objectweb.proactive.extensions.masterworker.interfaces.Task:

```
/**
 * Definition of a Task (to be executed by the framework) <br/>
 * @author The ProActive Team
 *
 * @param <R> the result type of this task
 */
@PublicAPI
public interface Task<R extends Serializable> extends Serializable {

    /**
     * A task to be executed<br/>
     * @param memory access to the worker memory
     * @return the result
     * @throws Exception any exception thrown by the task
     */
    R run(WorkerMemory memory) throws Exception;
}
```

Users need to implement the Task interface to define their tasks. The WorkerMemory parameter is explained in the Advanced Usage section.

Tasks are submitted to the master which in turn sends them to the workers. The following method submits the tasks:

```
public void solveIntern(final String originatorName,
    final List<? extends Task<? extends Serializable>> tasks) throws IsClearingError;
```

Warning

The master keeps a track of task objects that have been submitted to it and which are currently computing. Submitting twice the same task object without waiting for the result of the first computation will produce a TaskAlreadySubmittedException.

6.6. Results retrieval and reception order

Results are collected by the master when the calculations are complete.

There are two ways of waiting for the results. The user application can either wait until one or every result is available (the thread blocks until the results are available) or ask the master for result availability and continue until the results are finally available. In the second case, the application thread does not block while the results are computed.

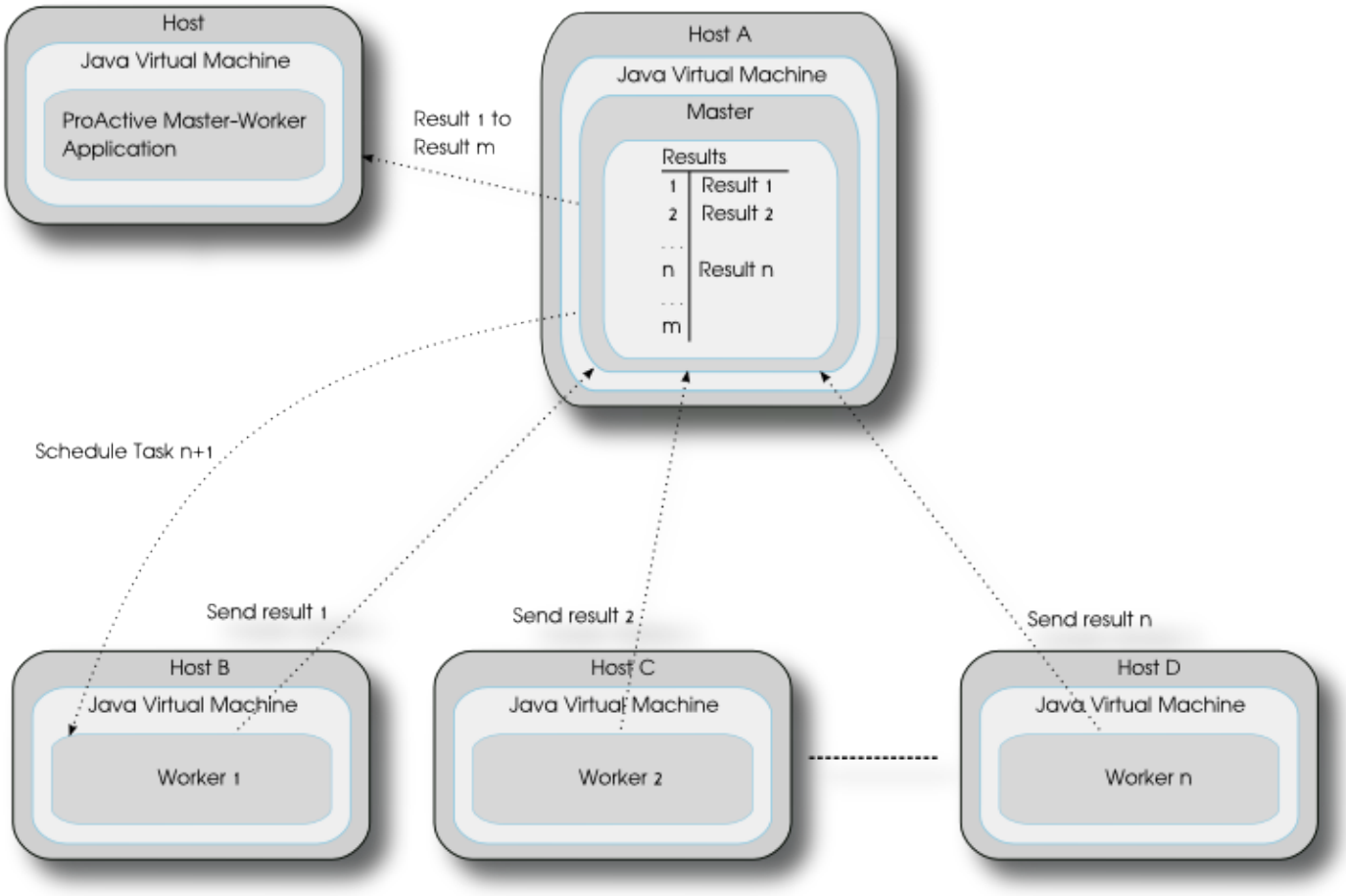


Figure 6.3. Results gathering

The results can be received in two modes:

- **Completion order mode (default)**: in this mode, user applications will receive the results in an unspecified order, depending on tasks completion order.
- **Submission order mode**: in this mode, user applications will receive the results in the same order as the task submitted to the master.

Result reception order can be switched from **Completion** order to **Submission** order using the following method:

```
/**
 * Sets the current ordering mode <br/>
 * If reception mode is switched while computations are in progress,<br/>
 * then subsequent calls to waitResults methods will be done according to the new mode.<br/>
 * @param mode the new mode for result gathering
 */
void setResultReceptionOrder(OrderingMode mode);
```

The default mode of the Master-Worker API is **Completion** order. The mode can be switched dynamically, which means that subsequent calls to wait methods (see below) will be done according to the new mode.

Five methods can be used in order to collect results:

- The first three methods will block the current thread until the corresponding results are available. If an exception occurs during the execution of one task, this exception will be thrown back to the user by the wait method.
- The fourth method will give information on results availability but will not block the user thread.
- The last method will tell when the user has received every result for the tasks previously submitted.

```
/**
 * Wait for all results, will block until all results are computed <br>
 * The ordering of the results depends on the result reception mode in use <br>
 * @return a collection of objects containing the result
 * @param originatorName name of the worker initiating the call
 * @throws org.objectweb.proactive.extensions.masterworker.TaskException if a task threw an Exception
 */
List<Serializable> waitAllResults(final String originatorName) throws TaskException, IsClearingError;

/**
 * Wait for the first result available <br>
 * Will block until at least one Result is available. <br>
 * Note that in SubmittedOrder mode, the method will block until the next result in submission order is available<br>
 * @param originatorName name of the worker initiating the call
 * @return an object containing the result
 * @throws TaskException if the task threw an Exception
 */
Serializable waitOneResult(final String originatorName) throws TaskException, IsClearingError;

/**
 * Wait for at least one result is available <br>
 * If there are more results availables at the time the request is executed, then every currently available results are
returned
 * Note that in SubmittedOrder mode, the method will block until the next result in submission order is available and will
return
 * as many successive results as possible<br>
 * @param originatorName name of the worker initiating the call
 * @return a collection of objects containing the results
```

```
 * @throws TaskException if the task threw an Exception
 */
List<Serializable> waitSomeResults(final String originatorName) throws TaskException;

/**
 * Wait for a number of results<br>
 * Will block until at least k results are available. <br>
 * The ordering of the results depends on the result reception mode in use <br>
 * @param k the number of results to wait for
 * @param originatorName name of the worker initiating the call
 * @return a collection of objects containing the results
 * @throws TaskException if the task threw an Exception
 */
List<Serializable> waitKResults(final String originatorName, int k) throws TaskException, IsClearingError;
```

6.7. Terminating the master

```
/**
 * Terminates the worker manager and (eventually free every resources) <br/>
 * @param freeResources tells if the Worker Manager should as well free the node resources
 */
void terminate(boolean freeResources);
```

One single method is used to terminate the master. A boolean parameter tells the master to free resources or not (i.e. terminate remote JVMs).

6.8. Worker ping period

At regular intervals, the Master sends a "ping" message to every Worker to check if they are reachable. The Ping period configuration parameter is the period in millisecond between two "ping" messages. The default value of this parameter is 10000 (which corresponds to 10 seconds).

In order to change this default value, the method described underneath can be called:

```
/**
 * Sets the period at which ping messages are sent to the workers <br/>
 * @param periodMillis the new ping period
 */
void setPingPeriod(long periodMillis);
```

If the Master does not receive an answer for the ping, then it will remove the Worker from its list and reassign the tasks the Worker has been assigned.

6.9. Worker task flooding

The Master-Worker API's internal scheduling mechanism is quite simple since it is based on a pulling strategy. When a worker has no more task to run, it asks the master for new tasks. The master usually gives a worker one task at a time, except the first time the worker asks for a task and each time the worker has no more tasks to compute. In this case, the master will do a **flooding**, by giving to worker as many tasks as the configurable parameter of the same name states. The default value of this parameter is 2, as it is expected to have at least twice as many tasks as workers. This mechanism is meant to avoid having idle workers waiting for new tasks all the time. The value of the flooding parameter should depend on how big your tasks are. A lot of small tasks should lead to a high flooding value (>10) where a small number of big tasks should lead to a small value (1-5).

Use the following method to change the flooding parameter:

```
/**
 * Sets the number of tasks initially sent to each worker
 * default is 2 tasks
 * @param number_of_tasks number of task to send
 */
void setInitialTaskFlooding(final int number_of_tasks);
```

6.10. Worker Memory

The Worker Memory purpose is to allow users to store and retrieve data from a **Worker's address space**. The typical use case is the Master-Worker API computation with an iterative process. An iterative process consists generally of an initialization step 0, followed by n computation steps, where step n needs the results of step n-1. The initialization steps often requires that a large amount of information is "loaded" into the worker. Without the worker memory access, this information would be lost at each step of the iteration, which means that the initialization step 0 needs to be done at step 1,2, ... n, etc...

The Worker Memory lets you send some initial memory content when workers are initialized. Later on, when tasks are executed, workers can have access to their memory and save or load data from it. Please note that this memory is not at all what is called a "shared memory". A shared memory would mean that the same memory would be shared by all workers. Here, each worker has its own private memory, and if a worker modifies its memory, the memory of other workers will not be affected.

6.10.1. Structure and API

The Worker memory structure is very simple: it consists of **<key, value>** associations. A Java object value is therefore saved in the memory with the given name, and this name will be needed to retrieve the value later on.

The Worker Memory API consists of three methods **save**, **load**, and **erase**. The interface to the worker memory is available when running a Task as a parameter of the run method. The user can use this interface to save, load or erase objects in the local worker's memory. Below is the detailed WorkerMemory interface:

```
/**
 * This interface gives access to the memory of a worker, a task can record data in this memory under a specific name.
<br/>
 * This data could be loaded later on by another task <br/>
 * @author The ProActive Team
 *
 */
@PublicAPI
public interface WorkerMemory {

  /**
   * Save data under a specific name
   * @param name name of the data
   * @param data data to be saved
   */
  void save(String name, Object data);

  /**
   * Load some data previously saved
   * @param name the name under which the data was saved
   * @return the data
   */
  Object load(String name);
```

```
    /**
     * Erase some data previously saved
     * @param name the name of the data which need to be erased
     */
    void erase(String name);
}
```

6.10.2. Storing data

A user can store data in the Workers' memory either when:

1. Workers are created remotely
2. A task is run on the Worker

Usage of the first mechanism is done by providing a list of <key, value> pairs (Map) to the constructors of the ProActiveMaster class. Every constructors detailed above have a version including this extra parameter. The given list will be the initial memory of every Workers created by the master.

Usage of the second mechanism is done by using the **WorkerMemory** parameter in the Task interface's **run** method. In contradiction with the first method, only the Worker currently running the Task will store the given data.

6.10.3. Retrieving and using the data

Loading and using any object stored in a Worker's memory is simply done through the **WorkerMemory** parameter in the **run** method of the **Task** interface.

6.11. Monte-Carlo PI Example

This very simple example computes PI using the Monte-Carlo method. The Monte-Carlo methods groups under the same name method which solves a problem by generating random numbers and examining how a fraction of the generated numbers follows certain patterns. The method can be used to obtain numerical results for problems which would be hard to solve through analytical methods. The complete example is available, along with more complex ones in the package org.objectweb.proactive.examples.masterworker

The task randomly creates a set of points belonging to the [0, 1[x[0, 1[interval and tests how many points are inside the unit circle. The number of points inside the unit circle allow us to calculate the value of PI with an arbitrary precision. The more points generated the better the accuracy for PI.

```
/**
 * Task which creates randomly a set of points belonging to the [0, 1[x[0, 1[ interval<br>
 * and tests how many points are inside the uniter circle.
 * @author The ProActive Team
 *
 */
public static class ComputePIMonteCarlo implements Task<Long> {

    /**
     *
     */
    public ComputePIMonteCarlo() {
    }

    public Long run(WorkerMemory memory) throws Exception {
        long remaining = NUMBER_OF_EXPERIENCES;
        long successes = 0;
```

```
        while (remaining > 0) {
            remaining--;
            if (experience()) {
                successes++;
            }
        }
        return successes;
    }

    public boolean experience() {
        double x = Math.random();
        double y = Math.random();
        return Math.hypot(x, y) < 1;
    }
}
```

In the main method, the master is created and resources are added using a deployment descriptor .

```
// creation of the master
ProActiveMaster<ComputePIMonteCarlo, Long> master = new ProActiveMaster<ComputePIMonteCarlo, Long>();

// adding resources
master.addResources(PIExample.class.getResource("MWApplication.xml"));
```

After the master is created, the tasks are created and submitted to the master.

```
// defining tasks
Vector<ComputePIMonteCarlo> tasks = new Vector<ComputePIMonteCarlo>();
for (int i = 0; i < NUMBER_OF_TASKS; i++) {
    tasks.add(new ComputePIMonteCarlo());
}

// adding tasks to the queue
master.solve(tasks);
```

After the task submission, the results are gathered and displayed.

```
// waiting for results
List<Long> successesList = master.waitAllResults();

// computing PI using the results
long sumSuccesses = 0;

for (long successes : successesList) {
    sumSuccesses += successes;
}

double pi = (4 * sumSuccesses) / ((double) NUMBER_OF_EXPERIENCES * NUMBER_OF_TASKS);

System.out.println("Computed PI by Monte-Carlo method : " + pi);
```

Finally, the master is terminated (all resources are freed) and the program exits.

```
master.terminate(true);
```

Chapter 7. The Calcium Skeleton Framework

7.1. Introduction

7.1.1. About Calcium

Calcium is part of the ProActive Grid Middleware for programming structured parallel and distributed applications. The framework provides a basic set of structured patterns (skeletons) that can be nested to represents more complex patterns. Skeletons are considered as a high level programming model because all the parallelisms details are hidden from the programmer. In Calcium, distributed programming is achieved by using ProActive deployment framework and active object model.

The Calcium implementation can be found in the following package of the ProActive Middleware distribution:

```
package org.objectweb.proactive.extensions.calcium;
```

7.1.2. The Big Picture

The following steps must be performed for programming with this framework.

1. Define the skeleton structure.
2. Implement the missing classes of the structure (the muscle codes).
3. Create a new Calcium instance.
4. Provide inputs of the problems to be solved by the framework.
5. Collect the results.
6. View the performance statistics.

Problems inputted into the framework are treated as tasks. The tasks are interpreted by the remote skeleton interpreters as shown in the following figure:

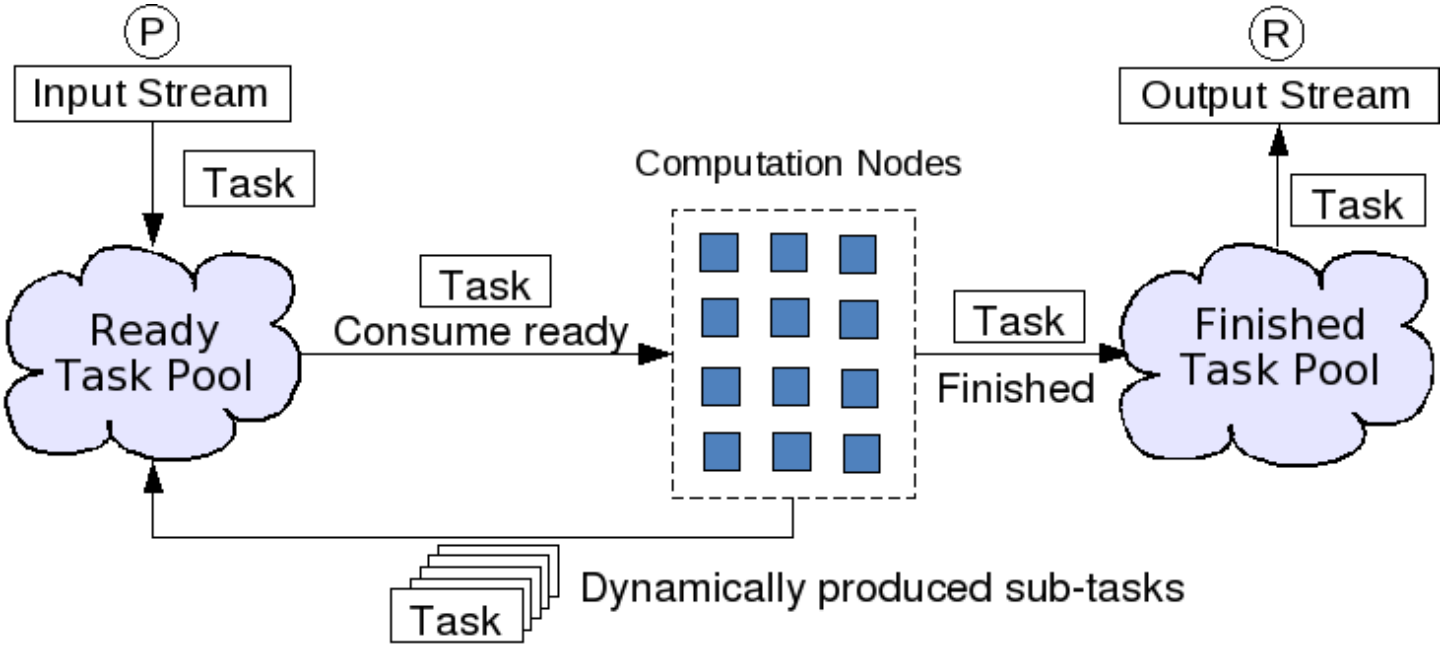


Figure 7.1. Task Flow in Calcium

All the generation, the distribution of tasks and their resources are completely hidden from the programmer. In fact, the task concept is never used when programming in Calcium.

7.2. Quick Example

Three examples are available in ProActive/src/Example/org/objectweb/proactive/examples/calcium and theirs launch scripts are located into ProActive/examples/calcium. In this section, we will deal with the FindPrimes example. We implement a skeleton program that finds prime numbers between an interval using a brute force (naive) approach.

7.2.1. Define the skeleton structure

The approach consists of dividing the original search space into several smaller search spaces, and computing each sub search space in parallel. Therefore, the most suitable pattern corresponds to Divide and Conquer.

```
public Skeleton<Interval, Primes> root;

public FindPrimes() {
   root = new DaC<Interval, Primes>(new IntervalDivide(), new IntervalDivideCondition(),
      new SearchInterval(), new JoinPrimes());
}
```

7.2.2. Implementing the Muscle Functions

We will call the problem an **Interval** and we will represent it using the following class.

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import java.io.Serializable;


class Interval implements Serializable {
   public int min;
   public int max;
   public int solvableSize;

   /**
    * Creates a new interval to search for primes.
    * @param min  Beginning of interval
    * @param max  End of interval
    * @param solvableSize Acceptable size of search interval
    */
   public Interval(int min, int max, int solvableSize) {
      this.min = min;
      this.max = max;
      this.solvableSize = solvableSize;
   }

}
```

The primes that are found will be stored in a **Primes** class.

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import java.io.Serializable;
import java.util.Vector;
```

```
public class Primes implements Serializable {
   public Vector<Integer> primes;

   public Primes() {
      primes = new Vector<Integer>();
   }
}
```

7.2.2.1. Divide

The division of an **Interval** into smaller intervals is handled by an **IntervalDivide** class:

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import org.objectweb.proactive.extensions.calcium.muscle.Divide;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;


public class IntervalDivide implements Divide<Interval, Interval> {

   public Interval[] divide(Interval param, SkeletonSystem system) {

      int middle = param.min + ((param.max - param.min) / 2);

      Interval top = new Interval(middle + 1, param.max, param.solvableSize);

      Interval bottom = new Interval(param.min, middle, param.solvableSize);

      return new Interval[] { top, bottom };
   }
}
```

7.2.2.2. Condition

The class **IntervalDivideCondition** is used to determine whether an **Interval** has to be subdivided or not.

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import org.objectweb.proactive.extensions.calcium.muscle.Condition;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;


public class IntervalDivideCondition implements Condition<Interval> {
   public boolean condition(Interval params, SkeletonSystem system) {
      return (params.max - params.min) > params.solvableSize;
   }
}
```

7.2.2.3. Skeleton

The **SearchInterval** class performs the actual finding of primes. This class will receive an **Interval** object and return a **Primes** object.

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import org.objectweb.proactive.extensions.calcium.muscle.Execute;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;
```

```
public class SearchInterval implements Execute<Interval, Primes> {
   public Primes execute(Interval param, SkeletonSystem system) {
      Primes primes = new Primes();

      for (int i = param.min; i <= param.max; i++) {
         if (isPrime(i)) {
            primes.primes.add(new Integer(i));
         }
      }

      return primes;
   }

   private boolean isPrime(int p) {
      for (int i = 2; i < p; i++) {
         if ((p % i) == 0) {
            return false;
         }
      }
      return true;
   }
}
```

7.2.2.4. Conquer

The **JoinPrimes** class consolidates the result of the sub intervals into a single result.

```
package org.objectweb.proactive.extensions.calcium.examples.findprimes;

import java.util.Collections;

import org.objectweb.proactive.extensions.calcium.muscle.Conquer;
import org.objectweb.proactive.extensions.calcium.system.SkeletonSystem;


public class JoinPrimes implements Conquer<Primes, Primes> {
   public Primes conquer(Primes[] p, SkeletonSystem system) {
      Primes conquered = new Primes();

      for (Primes param : p) {
         conquered.primes.addAll(param.primes);
      }

      Collections.sort(conquered.primes);
      return conquered;
   }
}
```

7.2.3. Create a new Calcium Instance

The instantiation of the framework is performed in the following way:

```
Environment environment = EnvironmentFactory.newMultiThreadedEnvironment(2);
```

```
Calcium calcium = new Calcium(environment);

Stream<Interval, Primes> stream = calcium.newStream(root);
```

7.2.4. Provide an input of problems to be solved by the framework

```
Vector<CalFuture<Primes>> futures = new Vector<CalFuture<Primes>>(3);
futures.add(stream.input(new Interval(1, 6400, 300)));
futures.add(stream.input(new Interval(1, 100, 20)));
futures.add(stream.input(new Interval(1, 640, 64)));
calcium.boot(); //begin the evaluation
```

7.2.5. Collect the results

```
for (CalFuture<Primes> future : futures) {
   Primes res = future.get();
   for (Integer i : res.primes) {
      System.out.print(i + " ");
   }
}
```

7.2.6. View the performance statistics

```
Stats futureStats = future.getStats();
System.out.println(futureStats);
```

```
StatsGlobal stats = calcium.getStatsGlobal();
System.out.println(stats);
```

7.2.7. Release the resources

```
calcium.shutdown();
```

7.3. Supported Patterns

Each skeleton represents a different parallelism described as follows:

- **Farm**: also known as **Master-Slave**, corresponds to the task replication pattern where a specific function must be executed over a set of slaves.
- **Pipe**: corresponds to computation divided in stages were the stage n+1 is always executed after the n-th stage.
- **If**: corresponds to a decision pattern, were a choice must be made between executing two functions.
- **While**: corresponds to a pattern were a function is executed while a condition is met.
- **For**: corresponds to a pattern were a function is executed a specific number of times.
- **Divide and Conquer**: corresponds to a pattern were a problem is divided into several smaller problems while a condition is met. The tasks are solved and then solutions are then conquered into a single final solution for the original problem.
- **Map**: corresponds to a pattern were the same function is applied to several parts of a problem: single instruction multiple data.
- **Fork**: is like map, but models multiple data multiple instruction.
- **Seq**: is used to wrap muscle functions into terminal skeletons.

Skeletons can be composed in the following way:

```
S := Farm(S)|Pipe(S1,S2)|If(cond,S1,S2)|While(cond,S)|For(i,S)|DaC(cond,div,S,conq)|Map(div, S, conq)|Fork(div,
 S1...SN, conq)|Seq(f)
```

For instance, you can use an **If** pattern to decide, depending on the value of **cond**, which pattern you want to use then (for example, if **cond** is true, then we will use a **While** pattern, otherwise, we will use a **For** one).

The Skeleton's API is the following:

```
class Farm<P,R> implements Skeleton<P,R>   { public Farm(Skeleton<P,R> child); }
class Pipe<P,R> implements Skeleton<P,R>   { <X> Pipe(Skeleton<P,X> s1, Skeleton<X,R> s2); }
class If<P,R> implements Skeleton<P,R>   { public If(Condition<P> cond, Skeleton<P,R> ifsub, Skeleton<P,R>
 elsesub); }
class While<P> implements Skeleton,P>   { public While(Condition<P> cond, Skeleton<P,P> child); }
class For<P> implements Skeleton<P,P>   { public For(int times, Skeleton,P> sub); }
class Map<P,R> implements Skeleton<P,R>    { public <X,Y> Map(Divide<P,X> div, Skeleton<X,Y> sub,
 Conquer<Y,R> conq); }
class Fork<P,R> implements Skeleton<P,R>   { public <X,Y> Fork(Divide<P,X> div, Skeleton<X,Y>... args,
 Conquer<Y,R> conq); }
class DaC<P,R> implements Skeleton<P,R>   { public DaC(Divide<P,P> div, Condition<P> cond, Skeleton<P,R> sub,
 Conquer<R,R> conq); }
class Seq<P,R> implements Skeleton<P,R>   { public Seq(Execute<P,R> secCode); }
```

Where the muscle functions are user provided, and have to implement the following interfaces:

```
interface Execute<P,R>  extends Muscle<P,R>       { public R exec(P param); }
interface Condition<P>  extends Muscle<P,Boolean> { public boolean evalCondition(P param); }
interface Divide<P,X>   extends Muscle<P,X[]>     { public List<X> divide(P param); }
interface Conquer<Y,R>  extends Muscle<Y[],R>     { public R conquer(Y[] param); }
```

7.4. Execution Environments

Calcium can be used with different execution environments, that is, an application implemented in one environment is able to run on a different one. There are currently 3 supported environments: **MultiThreadedEnvironment** (stable), **ProActiveEnvironment** (stable), **ProActiveSchedulerEnvironment** (beta).

7.4.1. MultiThreadedEnvironment

The **MultithreadEnvironment** is the simplest execution environment. It uses threads to execute tasks, and can thus be used efficiently on multiprocessor machines. It is also an easier environment to debug applications, before submitting them to a distributed environment.

```
Environment environment = EnvironmentFactory.newMultiThreadedEnvironment(2);
```

7.4.2. ProActiveEnvironment

The ProActiveEnvironment is the current stable way of executing a skeleton program on a distributed, but controlled, execution environment. It is mostly suitable for short lived distributed applications, as it does not yet support a suitable fault-tolerance mechanism. The ProActiveEnvironment uses ProActive Deployment Descriptors to acquire computation nodes and it uses active objects to communicate and distribute the program to the computation nodes.

```
String descriptor = FindPrimes.class.getResource("LocalDescriptor.xml").getPath();
Environment environment = EnvironmentFactory.newProActiveEnvironment(descriptor);
```

To instantiate the environment, a path toward a descriptor deployment has to be specified. The ProActiveEnvironment requires that this descriptor file provides the following contractual variables:

```
<variables>
    <descriptorVariable name="SKELETON_FRAMEWORK_VN" value="framework" />
   <descriptorVariable name="INTERPRETERS_VN" value="interpreters" />
    <programDefaultVariable name="MAX_CINTERPRETERS" value="3"/>
</variables>
```

Where the variables represent:

- **SKELETON_FRAMEWORK_VN**: The virtual-node pointing to the node where the service active object will be placed. This node should be stable and underloaded, since it will hold important objects like the TaskPool and the FileServer.
- **INTERPRETERS_VN**: The virtual-node pointing to the nodes that will be used for computation. It is important that this nodes can communicate with the nodes identified in SKELETON_FRAMEWORK_VN.
- **MAX_CINTERPRETERS**: (optional variable) Represents the number of maximum tasks that can be queued on an interpreter. By default, this value is **3**, but can be overridden using this variable.

The new GCM Deployment mechanism is also supported through the following factory mechanism:

```
String descriptor = FindPrimes.class.getResource("LocalDescriptor.xml").getPath();
Environment environment = EnvironmentFactory.newProActiveEnviromentWithGCMDeployment(descriptor);
```

7.4.3. ProActiveSchedulerEnvironment (unstable)

The ProActiveSchedulerEnvironment is suitable for executing long running applications, and uses the ProActive Scheduler at the lower level to handle the distribution and execution of tasks. Currently, tasks requiring file access and transfer are not supported using this environment, but will be supported in future releases.

```
Environment environment = ProActiveSchedulerEnvironment.factory("schedulerURL","user", "password");
```

To use the scheduler, a URL with its location, a username and a password must be provided.

The ProActiveSchedulerEnvironment is currently under development and as such represents an unstable version of the framework, thus it is located in the following package of the ProActive distribution:

```
package org.objectweb.proactive.extra.calcium.environment.proactivescheduler;
```

7.5. File Access and Transfer Support (beta)

Calcium provides a transparent support for file data access, based on the Proxy Pattern. The **BLAST** example is implemented using this support:

```
package org.objectweb.proactive.extensions.calcium.examples.blast;
```

The goal of the file transfer support is to minimize the intrusion of non-functional code inside muscle functions, such as code for moving downloading, uploading or moving data.

7.5.1. WorkSpace

The **workspace** is an abstraction that can be used to create files from inside muscle functions. The framework guarantees that: 1. Any **File** created in the workspace will have read/write permissions; 2. If a **File** is passed as parameter to other muscle functions, the **File** will be locally available when another muscle function access it. Where **File** corresponds to the standard Java type **java.io.File**.

```
package org.objectweb.proactive.extensions.calcium.system;

import java.io.File;
import java.io.FileFilter;
import java.io.IOException;
import java.net.URL;

import org.apache.log4j.Logger;
import org.objectweb.proactive.annotation.PublicAPI;
import org.objectweb.proactive.core.util.log.Loggers;
import org.objectweb.proactive.core.util.log.ProActiveLogger;
```

```
/**
 * This class is the interface for creating files on the computation node.
 *
 * This class is not Serializable on purpose, since it is environment dependent.
 *
 * @author The ProActive Team
 */
@PublicAPI
public interface WSpace {
   static Logger logger = ProActiveLogger.getLogger(Loggers.SKELETONS_SYSTEM);

   /**
    * Copies a File into the workspace, and returns
    * a reference on the new File.
    *
    * @param src  The original location of the file.
    * @return A reference to the file inside the workspace.
    * @throws IOException
    */
   public File copyInto(File src) throws IOException;

   /**
    * Downloads the file specified by the URL and places a copy inside
    * the workspace.
    *
    * @param src The location of the original file
    * @return A reference to the file inside the workspace.
    * @throws IOException
    */
   public File copyInto(URL src) throws IOException;

   /**
    * This method is used to get a reference on a file inside the workspace.
    *
    * Note that this is only a reference, and can point to an unexistent File.
    * ie. no File is actually created by invoking this method.
    *
    * @param path The path to look inside the workspace.
    * @return A reference to the path inside the workspace.
    */
   public File newFile(String path);

   /**
    * This method is used to get a reference on a file inside the workspace.
    *
    * Note that this is only a reference, and can point to an unexistent File.
    * ie. no File is actually created by invoking this method.
    *
    * @param path The path to look inside the workspace.
    * @return A reference to the path inside the workspace.
    */
   public File newFile(File path);
```

```
    /**
     * This method returns true if a file with this name exists in the work space.
     *
     * @return true if the file exists, false otherwise.
     */
    public boolean exists(File path);

    /**
     * This method returns a list of the files currently available on the root
     * of the workspace.
     *
     * @return The list of files that are currently held in the workspace.
     */
    public File[] listFiles();

    /**
     * This method returns a list of the files currently available on the root
     * of the workspace that match the specified filter.
     *
     * @param filter The filter to use for listing the files.
     * @return The list of files thar are currently held in the workspace and match the specified filter.
     */
    public File[] listFiles(FileFilter filter);
}
```

Files are treated in a deep-copy fashion, analogous with parameters/results of muscle functions. That is to say, when a **File** reference is passed from one muscle function to the next, the **File**'s data is copied. From this point modifications made on on the **File** by different muscle functions are made on copies of the **File**'s data.

7.5.2. Annotated Muscle Functions

Muscle functions can be annotated to improve the file transfer performance. By annotating a muscle function, the Calcium framework will try to pre-fetch files matching the annotation, and passed as parameters to the function in advance, before the function is executed. The current supported annotation can fetch a file based on its name and size:

```
public @interface PrefetchFilesMatching {
    String name() default "[unassigned]";

    long sizeSmallerThan() default Long.MIN_VALUE;

    long sizeBiggerThan() default Long.MAX_VALUE;
}
```

It is important to note that annotations only represent an optimization and are not required for the **File** support to work.

7.5.3. Muscle Function Example

The following example is taken from the BLAST skeleton program. The muscle function presented here conquers the results of several parallel BLASTs into a single file. First, the annotation is used to try and pre-fetch files that begging with the prefix "merged". Then, a new **File** is created in the workspace to hold the merged files. Then, a mergeFiles function is called to merge the results, and is not detailed here since it is specific to BLAST. Finally the new **File** holding the merged results is returned.

```
@PrefetchFilesMatching(name = "merged.*")
public class ConquerResults implements Conquer<File, File> {
    static Logger logger = ProActiveLogger.getLogger(Loggers.SKELETONS_APPLICATION);
```

```
  public File conquer(File[] param, SkeletonSystem system) throws Exception {
    if (logger.isDebugEnabled()) {
      logger.debug("Conquering Results");
    }

    WSpace wspace = system.getWorkingSpace();

    //Create a reference on the result merged file
    File merged = wspace.newFile("merged.result" + ProActiveRandom.nextPosInt());

    //Merge the files
    mergeFiles(merged, param);

    //Return the result
    return merged;
  }

  /**
   * This method takes a list of files and copies their content into a new
   * file, merging them all together.
   *
   * @param merged
   *          The output file, which will hold the merged result.
   * @param files
   *          The list of files to merge.
   *
   * @throws IOException
   */
  private void mergeFiles(File merged, File[] files) throws IOException {
    if (logger.isDebugEnabled()) {
      logger.debug("Conquering results into File: " + merged);
    }

    BufferedWriter bw = new BufferedWriter(new FileWriter(merged));
    for (File f : files) {
      BufferedReader br = new BufferedReader(new FileReader(f));

      String line = br.readLine();
      while (line != null) {
        bw.write(line + System.getProperty("line.separator"));
        line = br.readLine();
      }
      br.close();
    }

    bw.close();
  }
}
```

7.5.4. Input and output files from the framework

The input and output are transparent as long as the **File** type class is used to reference files. All File type referenced from inside the input parameter are imported into the Calcium framework when submitting the parameter into the stream.

```
Stream<BlastParams, File> stream = calcium.newStream(root);
CalFuture<File> future = stream.input(parameters);

try {
   File res = future.get();
} catch (Exception e) {
   e.printStackTrace();
}
```

Then, when the result is available, all **File** typed referenced in the result object are copied into the local machine before the result is returned to the user.

7.6. Performance Statistics (beta)

There are two types of performance statistics.

7.6.1. Global Statistics

These statistics refer to the global state of the framework by providing state information. The tasks can be in three different states: **ready** for execution, **processing** , **waiting** for other tasks to finish, and **finished** (ready to be collected by the user). The statistics corresponding to these states are:

- Number of tasks on each state.
- Average time spent by the tasks on each state.

Statistics for a specific moment can be directly retrieved from the Calcium instance:

```
StatsGlobal stats = calcium.getStatsGlobal();
```

An alternative is to create a monitor that can perform functions based on the statistics. In the following example, we activate a simple logger monitor that prints the statistics every second.

```
Monitor monitor = new SimpleLogMonitor(calcium, 1);
monitor.start();
// ...
monitor.stop();
```

7.6.2. Local Result Statistics

This statistics are specific for each result obtained from the framework. They provide information on how the result was obtained:

- Execution time for each muscle of the skeleton.
- Time spent by this task in the **ready**, **processing**, **waiting** and **executing** state. Also, the wallclock and computation time are provided.
- Data parallelism achieved: tree size, tree depth, number of elements in the tree.

```
for (CalFuture<Primes> future : futures) {
   Primes res = future.get();
   Stats futureStats = future.getStats();
   System.out.println(futureStats);
}
```

Chapter 8. OOSPMD

8.1. OOSPMD: Introduction

SPMD stands for Single Program Multiple Data, which is a technique used to parallelize applications by separating tasks and running them simultaneously on different machines or processors. ProActive allows the use of object oriented programming combined with the SPMD techniques.

ProActive uses group communication with SPMD in order to free the programmer from having to implement the complex communication code required for setting identical groups in each SPMD activity. Group communication allows the focus to be on the application itself and not on the synchronizations. An SPMD group is a group of active objects where each one has a group referencing all the active objects.

This chapter 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 a better structuring flexibility and implementation through a modest size API. The automation of key communication mechanisms and synchronization simplifies code writing.

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 makes possible for one activity to call methods on the group.

8.2. SPMD Group Creation

The main class for the SPMD groups is org.objectweb.proactive.api.PASPMD. This class contains methods for creating and controlling ProActive SPMD groups.

An SPMD group is a ProActive typed group built with the PASPMD.newSPMDGroup() method. This method looks like the PAGroup.newGroup() method: they have similar behavior (and overloads). The difference is that each member of an SPMD group has a reference to a group containing all the others members and itself (i.e. a reference to the SPMD group itself).

For a standard Java class:

```
public class A implements Active, java.io.Serializable {

   private String name;
   private int fooCounter = 0;
   private int barCounter = 0;
   private int geeCounter = 0;
   private String errors = "";

   public A() {
   }

}
```

The SPMD group is built as follows:

```
Object[][] params = { { "Agent0" }, { "Agent1" }, { "Agent2" } };
Node[] nodes = { NodeFactory.getDefaultNode(), super.getANode(), super.getANode() };
this.spmdgroup = (A) PASPMD.newSPMDGroup(A.class.getName(), params, nodes);
```

Object members of an SPMD group are aware about the whole group. They can obtain some information 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 information or to

communicate with method invocations. Those information are respectively obtained using the following static methods of the PASPMD class.

```
public static int getMySPMDGroupSize()
```

```
public static int getMyRank()
```

```
public static Object getSPMDGroup()
```

8.3. Synchronizing activities with barriers

Synchronizing processes is an important operation in any distributed system. Managing concurrency when several processes try to reach the same resource and coordinating the action of several processes to perform a joined computation are important parts of distributed computing.

There are different techniques to synchronize processes. With a SPMD application, we have the **synchronization barriers** mechanism.

Traditional barriers, as those provided for example by MPI, have a blocking behavior. When a process meets such a barrier, it fully stops its execution immediately and informs all the other members of the SPMD group about its state. It is only when all the other members have reach this barrier too that all the processes will be able to continue their execution. As the barriers are often used to synchronize the communication between processes, it is inefficient to completely stop the execution of a process without distinction between local computation and communication.

As a solution to this inefficiency, ProActive provides non-blocking barriers to only synchronize communication between activities. When an activity reaches a barrier instruction while executing a service, it will continue the execution of the current service until its end and tag all the next outgoing requests as post barrier. Consequently, any activity receiving a request tagged like this will delay its service until all the activities implied in this synchronization are ready to synchronize. An example of this behavior is illustrated on the Figure 8.1, “Behaviour example of a total barrier”.

This example presents the timelines of three active objects which are performing one type of barrier called a **total barrier**. The requests A and B are served before the synchronization as they were sent before the reach of the barrier instruction (symbolized by a red point). On the contrary, the requests **X** and **Y** are served **after** the synchronization, as they were sent **after** the barrier instruction.

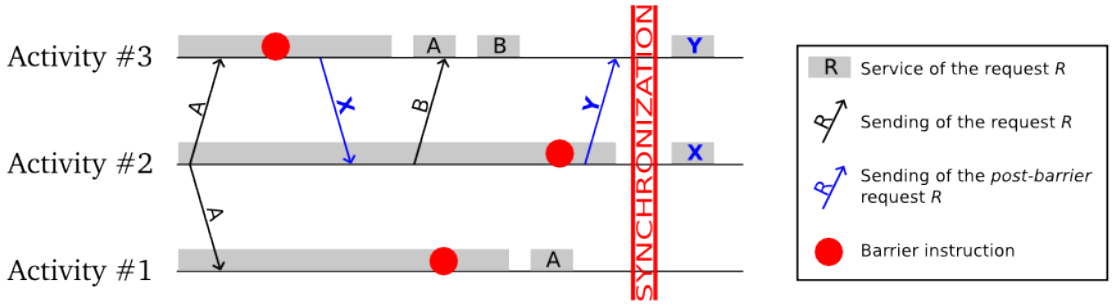


Figure 8.1. Behaviour example of a total barrier

The OOSPMD programming model provided by ProActive offers three kinds of barriers to synchronize activities:

- the **Total Barrier**
- the **Neighbour Barrier**
- the **Method-Based Barrier**

8.3.1. Total Barrier

Total barrier directly involves the SPMD group. A call to

```
public static void totalBarrier(String barrierName)
```

will block until all the members in the SPMD group have reached and called the identical barrier primitive. Such 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).

Warning

The string parameter is used as a 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**:

```
myspmdgroup.foo();
PASPMD.totalBarrier("'1'");
myspmdgroup.bar();
PASPMD.totalBarrier("'2'");
myspmdgroup.gee();
```

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 follows:

```
Object[][] params = { { "Agent0" }, { "Agent1" }, { "Agent2" } };
Node[] nodes = { NodeFactory.getDefaultNode(), super.getANode(), super.getANode() };
this.spmdgroup = (A) PASPMD.newSPMDGroup(A.class.getName(), params, nodes);
```

When the method start() is called on different instances of the class containing it, the different instances will wait for myspmdgroup.foo() to be completed and the barrier call to be made before they start myspmdgroup.bar(). The instances will also synchronize after the call to myspmdgroup.bar() as there is also a second barrier PASPMD.totalBarrier("'2'").

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

8.3.2. Neighbour barrier

The Neighbour barrier is a kind of lightweight barrier, involving only the Active Objects specified in a given group.

neighbourBarrier(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 **neighbours** as they are usually local to a given topology. An object that invokes the Neighbour barrier **HAVE TO BE IN THE GROUP** given as parameter. The **barrier message** is only sent to the group of neighbours.

The programmer has to explicitly build this group of neighbours. The topology API can be used to build such a 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 neighbour members (**left**, **right**, **up**, **down**). See the javadoc of the topology package for more information (org.objectweb.proactive.core.group.topology).

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

```
// synchronization to be sure that all submatrix have exchanged borders
PASPMD.neighbourBarrier("SynchronizationWithNeighbors" + this.iterationsToStop, this.neighbors);
```

Refer to the **Jacobi** example to see a use-case of the Neighbour barrier. Each submatrix needs only to be synchronized with the submatrices which are in its cardinal neighbours.

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

8.3.3. Method Barrier

The Method barrier does not involve more 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.

As a standard barrier, the method methodBarrier(String[]) will finish the current request served by the active object that calls it, but it then waits 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:

```
PASPMD.methodBarrier({'foo', 'bar', 'gee'});
```

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

```
PASPMD.methodBarrier({'foo'});
PASPMD.methodBarrier({'bar'});
PASPMD.methodBarrier({'gee'});
```

A method barrier is used without any group (SPMD or not). To learn more on Groups, please refer to Chapter 3, *Typed Group Communication*.

8.4. MPI to ProActive Summary

MPI	**ProActive**
MPI_Init and MPI_Finalize	Activities creation
MPI_Comm_Size	PASPMD.getMyGroupSize
MPI_Comm_Rank	PASPMD.getMyRank
MPI_Send and MPI_Recv	Method call
MPI_Barrier	PASPMD.barrier
MPI_Bcast	Method call on a group
MPI_Scatter	Method call with a scatter group as parameter
MPI_Gather	Result of a group communication
MPI_Reduce	Programmer's method

Table 8.1. MPI to ProActive

Chapter 9. Wrapping Native MPI Application

The **Message Passing Interface (MPI)** is a widely adopted communication library for parallel and distributed computing. This chapter explains how Proactive can help you to **deploy**, **wrap** and **couple** MPI applications for distributed environments

Note

Please note that, for the moment, the wrapping of native MPI applications is only available for *nix Operating Systems

Three different MPI wrapping approaches are possible:

- **Simple deployment of unmodified MPI applications (Section 9.1, “Simple Deployment of unmodified MPI applications”)**

  You will follow this approach if you want to take profit from ProActive GCM Deployment to handle deployment issues (access, allocation and remote execution) to deploy your existing MPI applications.
- **MPI code coupling (Section 9.2, “ MPI Code Coupling”)**

  This wrapping approach was designed to couple independent standalone MPI applications to make them interoperate, using a ProActive layer for inter-application communication.
- **MPI code wrapping (Section 9.3, “MPI Code Wrapping”)**

  This wrapping approach consists on supporting "MPI to/from Java" communications which permit users to exchange data between MPI and Java applications.

The MPI Native Applications Wrapping is organized along three different packages:

- org.objectweb.proactive.extensions.nativeinterface - contains the implementation of code wrapping (Java-C communication, data conversion and management)
- org.objectweb.proactive.extensions.nativeinterfacempi - contains the implementation of code coupling of MPI applications (including data conversion and MPI deployment management)
- org.objectweb.proactive.extensions.nativecode - contains a simple bootstrap class to launch wrapped distributed/parallel applications

9.1. Simple Deployment of unmodified MPI applications

As the name says, the simple deployment of MPI applications is intended to deploy standalone MPI applications. It does not require the usage of any specific API. It is just a matter of defining the application deployment in terms of GCMD and GCMA descriptors and loading/deploying these descriptors (please refer to Chapter 14, *ProActive Grid Component Model Deployment* to get more information about GCM Deployment). The advantage of deploying an MPI application with GCM deployment is that you can benefit from de support of various network protocols and grid/cloud middlewares to transparently deploy your MPI applications without configuring the environment or using specific tools.

The following descriptors are part of an application example, which can be deployed with the script HelloExecutableMPI.sh available in ProActive/examples/mpi/standalone_mpi.

Initially, the GCMA application has to be defined. The following snippet of code shows the definition of an MPI application. In that case, it is a simple application that implements a parallel “hello world” in MPI:

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">
```

```
<!-- Definition of environment constants-->
<environment>
   <javaPropertyVariable name="proactive.home"/>
   <javaPropertyVariable name="deployment.gcmd"/>
   <descriptorVariable name="cmd" value="${proactive.home}/examples/mpi/standalone_mpi/hello_mpi"/>
</environment>

<!-- Definition of the MPI command along with arguments and the resources
provider that will be responsible for the obtention/accesss to the
resources-->
<application>
   <mpi>
      <command name="${cmd} arg1 arg2 arg3"/>
      <nodeProvider refid="LAN_MPI" />
   </mpi>
</application>

<!--Resource provider (GCMD descriptor)-->
<resources>
   <nodeProvider id="provider">
      <file path="${deployment.gcmd}" />
   </nodeProvider>
</resources>

</GCMApplication>
```

The definition of node providers follows the philosophy of a Java application deployment. But, instead of using an SSH (or any other) group, the group is defined as an MPI group. The following snippet of descriptor shows a simple GCMD which defines the deployment of the application over a set of distributed resources:

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


   <environment>
      <javaPropertyVariable name="user.home" />
      <javaPropertyVariable name="proactive.home" />
   </environment>

   <resources>
      <group refid="LAN_MPI">
         <host refid="ComputeNode" />
      </group>
   </resources>

   <infrastructure>

      <hosts>
         <host id="ComputeNode" os="unix">
            <homeDirectory base="root" relpath="${user.home}" />
         </host>
```

```
    </hosts>

    <!--Definition of the MPI group-->
    <groups>
      <mpiGroup
        id="LAN_MPI"
        hostList="localhost localhost"
        distributionPath="/usr"
        commandPath="${proactive.home}/scripts/gcmdeployment/executable_mpi_mpich.sh">
      </mpiGroup>
    </groups>
  </infrastructure>
</GCMDeployment>
```

Please not that, on the definition of the MPI group, some different tags can be used. The following tags are mandatory:

- **hostlist** - list of resources where the MPI application will be deployed
- **distributionPath** - installation path of MPI. In the example shown above, MPI is supposed to be installed system-wide. So, the mpirun command can be found in /usr/bin
- **commandPath** - the deployment of MPI application by ProActive is actually done by scripts which are responsible for the configuration of the environment, before the execution. This is also the place where the runtime handles the different MPI distribution characteristics (e.g. LAM/MPI requires the creation of a virtual LAM environment).

  The ProActive middleware offers support for four different MPI distributions:
  - LAM/MPI: use the ProActive/scripts/gcmdeployment/executable_mpi_lam.sh commandPath
  - MPICH or OpenMPI: use the ProActive/scripts/gcmdeployment/executable_mpi_mpich.sh commandPath
  - GridMPI: use the ProActive/scripts/gcmdeployment/ProActive/scripts/gcmdeployment/executable_gridmpi.sh commandPath

  But you can use any other distribution by defining a similar scripts and using them on GCMD.

The following tags are optional on the definition of **MPIGroups**:

- **commandOptions** - list of MPI-related option to be appended on the mpirun command at deployment time
- **machineFile** - file containing a list of machines. If this file is not provided, the runtime automatically creates the machineFile, including the hostnames defined in the hostlist tag. If the file is generated at runtime, it is deleted prior to the end of the execution.
- **execDir** - directory where the mpirun command will be executed. This allows the use of relative path on the definition of the MPI binary on the GCMA descriptor.

Once you have defined the 2 descriptors (GCMA and GCMD), the deployment of the application is pretty straightforward:

```
import java.io.File;

import org.apache.log4j.Logger;
import org.objectweb.proactive.api.PALifeCycle;
import org.objectweb.proactive.core.util.log.Loggers;
import org.objectweb.proactive.core.util.log.ProActiveLogger;
import org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment;
import org.objectweb.proactive.gcmdeployment.GCMApplication;


public class HelloExecutableMPI implements java.io.Serializable {

  public static void main(String[] args) throws Exception {
```

```
        GCMApplication applicationDescriptor = PAGCMDeployment.loadApplicationDescriptor(new File(args[0]));
        applicationDescriptor.startDeployment();


        PALifeCycle.exitSuccess();
    }
}
```

9.2. MPI Code Coupling

This wrapping approach was designed to couple independent standalone MPI applications to make them interoperate, using a dedicated MPI-like communication API and seamless using a ProActive communication inter-application layer. Besides the ease of deployment, the MPI code coupling can be used to perform transparent inter-cluster communication, even if nodes do not present a direct link among them.

There are two basic usages for the ProActive/MPI coupling approach:

- **Single MPI application** - the idea of code coupling in the context of a single application is to make possible the execution of a single MPI application across a multi-domain environment, which could be composed of clusters, grids and/or cloud resources. This usage is specially intended for embarrassingly distributed/parallel applications or applications parallelized upon a domain-decomposition approach. For instance, you might have a large data mesh and intend to distribute more this mesh to avoid memory swapping and cash misses.
- **Multiple MPI applications**: this is a more loosely coupled approach which allows to loose-coupling multiple MPI applications which can be deployed in different clusters, grids and/or cloud resources. This usage is specially intended for distributed and parallel applications parallelized upon a functional-decomposition approach. For instance, in a typical weather modeling, you might have multiple independent numerical kernels which implement different models (hydrology, pressure and temperature, for instance), processing independent sets of data and explicit data dependencies among these kernels.

In both cases, the MPI-coupling scenario looks like the following picture:

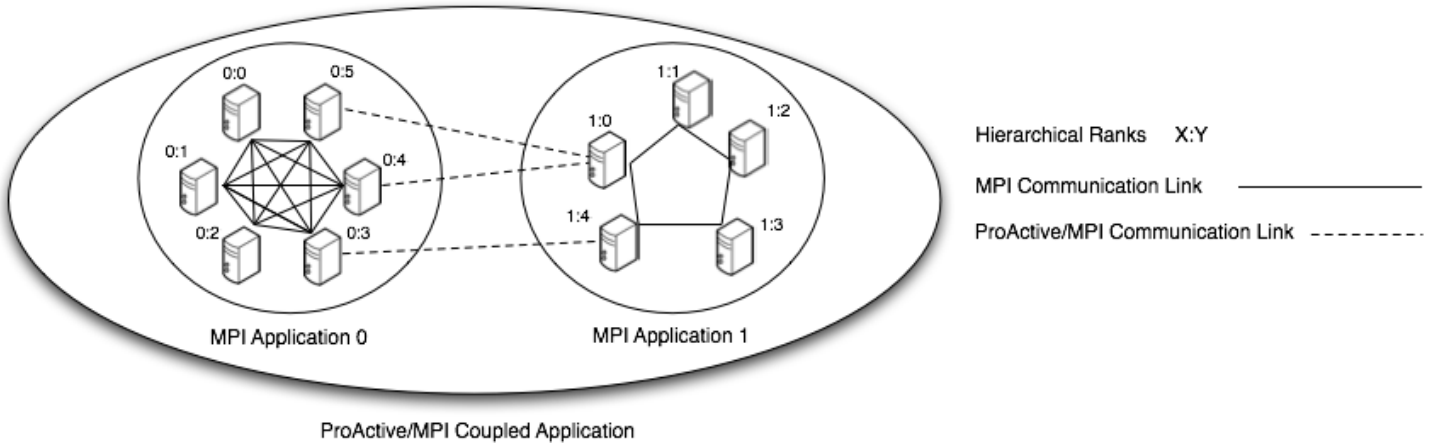


Figure 9.1. MPI-coupling scenario

On this picture, we can identify two independent MPI applications (Application 0 and Application 1) running on two sets of resources. Internally, each of these applications communicates through standard MPI and the two applications can communicate through the ProActive/MPI communication layer, due to an API defined on Section 9.2.1, “MPI Code Coupling API”.

Each of the processes receives a hierarchical identification X:Y, being X the application ID and Y the MPI rank. By using the API and the hierarchical ranks, the different applications can address process of different applications to communicate.

9.2.1. MPI Code Coupling API

The MPI Code Coupling API is a simplified MPI-like set of primitives, exposed through C/C++ and Fortran bindings. This API is intended to enable the explicit communication among the different MPI applications. Since ProActive plays just a middleware role in the case of code coupling, there is no Java API associated to code coupling. Section 9.3, “MPI Code Wrapping” presents an API that allows Native-Java communication.

```
#include "ProActiveMPI.h"

/**
 * ProActiveMPI_Init
 * Init ProActiveMPI environment, bind the native process to the Java wrapping process.
 *
 * Input parameters
 * rank    - MPI rank of the process
 */
int ProActiveMPI_Init(int rank);

/**
 * ProActiveMPI_Finalize
 * Finalize ProActiveMPI infrastructure.
 */
int ProActiveMPI_Finalize();

/**
 * ProActiveMPI_Job
 * Finalize ProActiveMPI infrastructure.
 *
 * Output parameters
 * job           - the number of coupled independent MPI applications
 * nb_process    - the global number of processes involved in the coupled application
 */
int ProActiveMPI_Job(int * job, int * nb_process);


/**
 * ProActiveMPI_Send
 * Sends data to a remote process, possibly running on a different MPI application.
 *
 * Input Parameters
 * buf      - the buffer to be sent
 * count    - number of elements in send buffer (nonnegative integer)
 * datatype - datatype of each recv buffer element
 * dest     - rank of destination (integer)
 * tag      - message tag (integer)
 * jobID    - remote job (integer)
 * request    - communication request (handle)
 */
int ProActiveMPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, int idjob );

/**
 * ProActiveMPI_Recv
 * Performs a blocking receive waiting from data from ProActive java side (eventually dispatched from a distant MPI
process)
```

```
 *
 * Output Parameters
 * buf    - initial address of receive buffer
 *
 * Input Parameters
 * count    - number of elements in send buffer (nonnegative integer)
 * datatype    - datatype of each recv buffer element
 * src        - rank of source (integer)
 * tag        - message tag (integer)
 * jobID    - remote job (integer)
 */
int ProActiveMPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag, int jobID);


/**
 * ProActiveMPI_IRecv
 * Performs a non blocking receive from MPI side to receive data from ProActive java side (eventually dispatched from a
 distant MPI process)
 *
 * Output Parameters
 * request    - communication request (handle)
 *
 * Input Parameters
 * buf        - initial address of receive buffer
 * count    - number of elements in send buffer (nonnegative integer)
 * datatype    - datatype of each recv buffer element
 * src        - rank of source (integer)
 * tag        - message tag (integer)
 * jobID    - remote job (integer)
 */
int ProActiveMPI_IRecv(void* buf, int count, MPI_Datatype datatype, int src, int tag, int idjob, ProActiveMPI_Request *
 request);


/**
 * ProActiveMPI_Bcast
 * Performs a broadcast to an group of processe
 *
 * Input Parameters
 * sendbuf            - initial address of sender buffer
 * count            - number of elements in send buffer (nonnegative integer)
 * datatype            - datatype of each recv buffer element
 * nb_send            - the ID of the sender
 * tag                - message tag (integer)
 * jobID            - remote job (integer)
 * pa_rank_array    - array of destinations
 */
int ProActiveMPI_Bcast(void * sendbuf, int count, MPI_Datatype datatype, int tag, int nb_send, int * pa_rank_array);

/**
 * ProActiveMPI_Scatter
 * Scatter a set of buffers to a set of destinations
 *
 * Input Parameters
```

```
 * buffers           - array of pointers to buffers
 * count           - array of integer containing number of elements in each send buffer (nonnegative integers)
 * datatype          - datatype of each recv buffer element
 * tag             - message tag (integer)
 * pa_rank_array    - array of destinations
 */
int ProActiveMPI_Scatter(int nb_send, void ** buffers, int * count_array, MPI_Datatype datatype, int tag, int *
 pa_rank_array);

/**
 * ProActiveMPI_Test
 * Tests for the completion of receive from a ProActive java class
 *
 * Output Parameters
 * flag    - true if operation completed (logical)
 *
 * Input Parameters
 * request    - communication request (handle)
 */
int ProActiveMPI_Test(ProActiveMPI_Request *request, int *flag);

/**
 * ProActiveMPI_Wait
 * Waits for an MPI receive from a ProActive java class to complete
 *
 * Input Parameters
 * request    - communication request (handle)
 */
int ProActiveMPI_Wait(ProActiveMPI_Request *request);
```

9.2.2. MPI Code Coupling Deployment and Example

This section presents an example of code coupling available in ProActive/src/Examples/org/objectweb/proactive/examples/mpi/proactive_mpi/. The following code is the MPI implementation of a small benchmark of the point-to-point synchronous and asynchronous communication between processes in two different MPI applications:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
/*Include ProActiveMPI header*/
#include "ProActiveMPI.h"


int main (int argc, char **argv)
{
  int myjobid, otherjobid, i, last, nprocs, allprocs, error, rank, size;
  double t0, t1, time;
  double *a, *b;
  double max_rate = 0.0, min_latency = 10e6;
  ProActiveMPI_Request request;

  a = (double *) malloc (1048576* sizeof (double));
  b = (double *) malloc (1048576* sizeof (double));
```

```
for (i = 0; i < 1048576; i++) {
   a[i] = (double) i;
   b[i] = 0.0;
}

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/*Init ProActiveMPI passing MPI rank as parameter*/
error = ProActiveMPI_Init(rank);
if (error < 0){
       printf("[MPI] !!! Error ProActiveMPI init \n");
       MPI_Abort( MPI_COMM_WORLD, 1 );
}

/*Obtain Application (or Job) ID*/
ProActiveMPI_Job (&myjobid, &allprocs);
otherjobid = (myjobid + 1) % 2;


/*Synchronous ProActiveMPI ping-pong*/
if (myjobid == 0) printf("\n  Synchronous ping-pong\n\n");

for (size = 8; size <= 8388608; size *= 2) {

   t0 = MPI_Wtime();
   if (myjobid == 0) {
      ProActiveMPI_Send(a, size/8, MPI_DOUBLE, 0, 0, otherjobid);
      ProActiveMPI_Recv(b, size/8, MPI_DOUBLE, 0, 0, otherjobid);
   } else {
      ProActiveMPI_Recv(b, size/8, MPI_DOUBLE, 0, 0, otherjobid);
      ProActiveMPI_Send(b, size/8, MPI_DOUBLE, 0, 0, otherjobid);
   }
   t1 = MPI_Wtime();
   time = 1.e6 * (t1 - t0);

      if (myjobid == 0 && time > 0.000001) {
       printf(" %7d bytes took %9.0f usec (%8.3f MB/sec)\n", size, time, 2.0 * size / time);

   } else if (myjobid == 0) {
          printf(" %7d bytes took less than the timer accuracy\n", size);
   }

}


/*asynchronous ProActiveMPI ping-pong*/
if (myjobid == 0) printf("\n Asynchronous ping-pong\n\n");

  for (size = 8; size <= 8388608; size *= 2) {


    ProActiveMPI_IRecv(b, size/8, MPI_DOUBLE, 0, 0, otherjobid, &request);
```

```
    t0 = MPI_Wtime();
    if (myjobid == 0) {
        ProActiveMPI_Send(a, size/8, MPI_DOUBLE, 0, 0, otherjobid);
        ProActiveMPI_Wait(&request);
    } else {
        ProActiveMPI_Wait(&request);
        ProActiveMPI_Send(b, size/8, MPI_DOUBLE, 0, 0,  otherjobid);
    }
    t1 = MPI_Wtime();
    time = 1.e6 * (t1 - t0);

      if (myjobid == 0 && time > 0.000001) {
       printf(" %7d bytes took %9.0f usec (%8.3f MB/sec)\n", size, time, 2.0 * size / time);

    } else if (myjobid == 0) {
          printf(" %7d bytes took less than the timer accuracy\n", size);
    }

  }

  /*Finalize ProActiveMPI application (before finalizing MPI application)*/
  ProActiveMPI_Finalize();
  MPI_Finalize();
}
```

In general, the structure of a ProActive-coupled MPI application application looks like a standard MPI application: it starts by an initial handshake to start the ProActive/MPI application (ProActiveMPI_Init) and obtain identifiers (MPI_Comm_rank and ProActiveMPI_Job). Then, it follows by the application core. This application core may contain the numerical kernels which involve communication and thus the use of MPI and ProActiveMPI applications. Eventually, it ends with the finalization of the application (ProActiveMPI_Finalize).

Note

ProActive/MPI applications have a communication semantic similar to MPI applications. This means that, despite the object-oriented implementation of the communication layer, communications are, in general, two-side operations (send/receive). The transport implementation in the context of the ProActive layer is completely asynchronous, but there is no relation between the asynchronous calls with futures and the Native-side of applications. For this reason, you should, as in MPI, prefer asynchronous communications over synchronous ones to improve applications performance.

Since this is a ProActive example, it can be simply compiled within the Ant target compile.extensions. Yet, you may need to compile/link your applications with ProActiveMPI native libraries in order to deploy your own applications. In order to do that, you will require the following steps:

1. First you must be under a *nix system and have MPI installed
2. Compile the compile.extensions Ant target to create native libraries
3. Include the folder ProActive/dist/lib/native (-LProActive/dist/lib/native)
4. Include the folders ProActive/classes/Extensions/org/objectweb/proactive/extensions/nativeinterface/ and ProActive/classes/Extensions/org/objectweb/proactive/extensions/nativeinterfacempi/control/config/src/ to the compilation (gcc / mpicc -I)
5. Include the libraries rt, lProActiveNativeInterfaceIPC and ProActiveMPIComm (-lProActiveNativeInterfaceIPC -lProActiveMPIComm).

Thus, your compilation command will look a bit like:

```
mpicc -O3 -lrt -I${ProActive}/classes/Extensions/org/objectweb/proactive/extensions/nativeinterfacempi/control/config/src/
 -I${ProActive}classes/Extensions/org/objectweb/proactive/extensions/nativeinterface/
 -L${ProActive}/dist/lib/native
 -lProActiveNativeInterfaceIPC
 -lProActiveMPIComm your_application.c -o your_application
```

Once you have your application compiled, you will have to write some deployment descriptors to deploy the java communication layer and each of the independent MPI applications.

The Java descriptor looks exactly like the descriptors presented in Chapter 14, *ProActive Grid Component Model Deployment*. The only extra requirement is the definition of a -Djava.library.path option which is the place where the Java runtime will load the native library that allows Java-C communication. This options has to point to ProActive/dist/lib/native.

ProActiveMPI GCMA:

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">

   <environment>
      <javaPropertyVariable name="proactive.home"/>
      <descriptorVariable name="descriptor.dir" value="${proactive.home}/examples/mpi/proactive_mpi"/>
      <descriptorVariable name="java.cmd"  value="java"/>
      <descriptorVariable name="proactive.properties" value="-Djava.library.path=${proactive.home}/dist/lib/
native"/>
      <descriptorVariable name="deployment.pa1.gcmd" value="${descriptor.dir}/gcmd.pa1.xml"/>
      <descriptorVariable name="deployment.pa2.gcmd" value="${descriptor.dir}/gcmd.pa2.xml"/>
   </environment>

   <application>
      <proactive base="root" relpath="${proactive.home}">
         <configuration>
            <java base="root" relpath="${java.cmd}" />
            <jvmarg value="${proactive.properties}" />
            <applicationClasspath>
               <pathElement base="proactive" relpath="dist/lib/ProActive_examples.jar"/>
               <pathElement base="proactive" relpath="dist/lib/ProActive.jar"/>
            </applicationClasspath>
            <securityPolicy base="proactive" relpath="examples/proactive.java.policy"/>
            <log4jProperties base="proactive" relpath="examples/proactive-log4j"/>
         </configuration>
         <virtualNode id="mpivn1" capacity="1">
            <nodeProvider refid="provider1" capacity="1"/>
         </virtualNode>
         <virtualNode id="mpivn2" capacity="1">
            <nodeProvider refid="provider2" capacity="1"/>
         </virtualNode>
      </proactive>
   </application>

   <resources>
```

```
    <nodeProvider id="provider1">
      <file path="${deployment.pa1.gcmd}" />
    </nodeProvider>
    <nodeProvider id="provider2">
      <file path="${deployment.pa2.gcmd}" />
    </nodeProvider>
  </resources>

</GCMApplication>
```

In the GCMA definition, you has to define as many resource providers as independent MPI applications.

Note

Pay attention that the resources used for the runtime deployment has to be the same used for MPI execution.

ProActiveMPI GCMD:

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

  <environment>
    <javaPropertyVariable name="user.home" />
    <javaPropertyVariable name="proactive.home" />
  </environment>

  <resources>
    <group refid="LAN_PA1">
      <host refid="ComputeNode" />
    </group>
  </resources>

  <infrastructure>
    <hosts>
      <host id="ComputeNode" os="unix" hostCapacity="1" vmCapacity="1">
        <homeDirectory base="root" relpath="${user.home}" />
      </host>
    </hosts>

    <groups>
      <sshGroup id="LAN_PA1" hostList="host1" />
    </groups>
  </infrastructure>
</GCMDeployment>
```

Note that we have defined the deployment of one runtime by JVM since the ProActiveMPI runtime are singleton and then, it is important to avoid runtime co-allocation.

The MPI deployment descriptors are exactly like the ones presented in Section 9.1, “Simple Deployment of unmodified MPI applications”.

MPI GCMA:

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">

  <environment>
  <javaPropertyVariable name="proactive.home"/>
  <descriptorVariable name="descriptor.dir" value="${proactive.home}/examples/mpi/proactive_mpi"/>
    <descriptorVariable name="deployment.gcmd" value="${descriptor.dir}/gcmd.mpi1.xml"/>
    <descriptorVariable name="mpi.cmd" value="${proactive.home}/classes/Examples/org/objectweb/proactive/
examples/mpi/proactive_mpi/hello_pampi"/>
    <descriptorVariable name="mpi.args" value="rien"/>
  </environment>

  <application>
    <mpi>
      <command name="${mpi.cmd} ${mpi.args}" />
      <nodeProvider refid="LAN_MPI" />
    </mpi>
  </application>

  <resources>
    <nodeProvider id="LAN_MPI">
      <file path="${deployment.gcmd}" />
    </nodeProvider>
  </resources>

</GCMApplication>
```

MPI GCMD:

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

  <environment>
    <javaPropertyVariable name="user.home" />
    <javaPropertyVariable name="proactive.home" />
    <descriptorVariable name="mpi.home" value="/usr/"/>
    <descriptorVariable name="mpi.cmd.path" value="${proactive.home}/scripts/gcmdeployment/
executable_mpi_gridmpi.sh"/>
    <descriptorVariable name="host.list" value="host1"/>
  </environment>

  <resources>
    <group refid="LAN_MPI1">
      <host refid="ComputeNode" />
    </group>
  </resources>

  <infrastructure>
    <hosts>
```

```
        <host id="ComputeNode" os="unix" hostCapacity="1" vmCapacity="1">
          <homeDirectory base="root" relpath="${user.home}" />
        </host>
      </hosts>

      <groups>
        <mpiGroup
          id="LAN_MPI1"
          hostList="${host.list}"
          distributionPath="${mpi.home}"
          commandPath="${mpi.cmd.path}">
        </mpiGroup>
      </groups>
    </infrastructure>
</GCMDeployment>
```

Once you have defined these descriptors, you can use the org.objectweb.proactive.extensions.nativecode.NativeStarter class. This class receives as parameters the Java ProActive GCMA and the MPI GCMAs:

```
org.objectweb.proactive.extensions.nativecode.NativeStarter gcma.pa.xml gcma.mpi1.xml gcma.mpi2.xml ...
```

9.2.3. The DiscoGrid Project

The DiscoGrid[1] project aims at studying and promoting a new paradigm for programming non-embarrassingly parallel scientific computing applications on distributed, heterogeneous, computing platforms. The target applications require the numerical resolution of systems of partial differential equations (PDEs) modeling electromagnetic wave propagation and fluid flow problems.

In the context of the DiscoGrid project, the ProActive team developed a GCM component-based runtime capable of coupling and deploying MPI applications over heterogeneous multi-domain infrastructures, composed by clusters, grids and clouds with the seamless treatment of complex network configurations (including firewalls and NAT).

If you are interested in more advanced use of the ProActive/MPI code coupling approach, you will probably be interested in the DiscoGrid project. Please refer to the the DiscoGrid forge[2] for documentation and download of the DiscoGrid distribution.

9.3. MPI Code Wrapping

This wrapping approach was designed to couple native applications and Java codes so as to make them interoperate by the usage of a simplified message passing-based API. This approach allows the development of native applications or numerical kernels to be bind to a native implementation.

Two scenarios motivate this wrapping approach:

- **Use of existing numerical kernels or applications**: Let us assume that you want to have part of your Java application to be handled by an existing native code to improve performance and/or to simply reuse these native kernels instead of translating them into Java. This is particularly useful for highly optimized native applications which can not be easily ported to Java.
- **Use of resources not available in Java (libraries, CUDA, etc.)**: unfortunately, a number of libraries are just available in other programming languages (such as C/C++ and Fortran). This is the case of numerous numerical libraries and libraries to handle hardware devices (for instance, CUDA computing engine).

In both cases, all you need is to perform a deployment of the native application (Section 14.7.2.1, “Executable”) and of the Java application (Chapter 14, *ProActive Grid Component Model Deployment*), and use an specific API to make them communicate. The compilation of the native application and its deployment follow exactly the same approach presented in Section 9.2, “ MPI Code Coupling”. The main difference relies on the use of your own Java application instead of the NativeStarter.

[1] http://www-sop.inria.fr/nachos/team_members/Stephane.Lanteri/DiscoGrid/
[2] https://gforge.inria.fr/projects/dg-proactive/

9.3.1. Code Wrapping API

The MPI Code Wrapping API is a simplified message passing based set of primitives, exposed on the native side through C/C++ bindings (Section 9.2.1, “MPI Code Coupling API”), and with a couple of API classes on the Java side.

9.3.1.1. Native Code Wrapping API

These are the main primitives available to init/finalize the binding of native and Java applications and communicate native-java processes:

```
#include "native_layer.h"

/**
 * init
 * Start the handshake that binds the native and Java application
 *
 * Input Parameters
 * creation_flag    - Since the implementation of the IPC communication is shared by Java/Native,
 *                    this flag identifies the origin of the function call
 *                    (if native side,  creation_flag = 0 else creation_flag = 1)
 */
int init(int creation_flag);

/**
 * terminate
 * Terminate the wrapping, releasing memory and cleaning up running environment
 */
int terminate();

/**
 * recv_message
 * Receive message from Java wrapper
 *
 * Output Parameters
 * lenght    - size in bytes of the buffer that will be received
 * data_ptr    - pointer to the byte buffer
 */
int recv_message(int * length, void ** data_ptr);

/**
 * recv_message_async
 * Receive message asynchronously from Java wrapper
 *
 * Output Parameters
 * lenght    - size in bytes of the buffer that will be received
 * data_ptr    - pointer to the byte buffer
 */
int recv_message_async(int * length, void ** data_ptr);

/**
 * send_message
 * Send message to Java wrapper
 *
 * Output Parameters
```

```
* lenght    - size in bytes of the buffer that will be received
* data_ptr    - pointer to data structure that indicated the message to be sent (message includes data type, signature and buffer pointer)
*/
int send_message(int length, void *data_ptr);
```

9.3.1.2. Java Code Wrapping API

The Java API is mainly offered through 2 classes:

- Communication and Code Coupling, on the org.objectweb.proactive.extensions.nativeinterface.coupling.NativeInterface[3] class.
- Data conversion, on the org.objectweb.proactive.extensions.nativeinterface.utils.ProActiveNativeUtil[4] class.

If you want a practical use of these APIs, please check Section 9.2, " MPI Code Coupling" and the Java and C implementations of the MPI Code coupling in the ProActive package org.objectweb.proactive.extensions.nativeinterfacempi.

[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../api_complete/org/objectweb/proactive/extensions/nativeinterface/coupling/NativeInterface.html

[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../api_complete/org/objectweb/proactive/extensions/nativeinterface/utils/ProActiveNativeUtil.html

Chapter 10. Accessing data with Data Spaces API

10.1. Introduction

In the ProActive library data can be accessed through Data Spaces API that is to be explained in the following chapter.

Data Spaces mechanism conceptually pose the virtual file system layer (further called VFS) between the user code, that accesses file system trough provided API (like Java IO, POSIX, etc.), and physical (local or remote) location of files. Therefore, a programmer does not need to be aware of real file location, and can use VFS abstraction instead. The Data Spaces mechanism brings such a point of view to the ProActive programmer with a dedicated API from the org.objectweb.proactive.extensions.dataspaces.api package.

In case of ProActive, Data Spaces API layers the file systems, that can be accessed through variety of access protocols, creating one virtual file system with uniform access among all Active Objects. This is achieved by ensuring that every Active Object has the same view of VFS tree, an URI used for accessing data in one Active Object is correct just after passing it to the others. Therefore sharing data becomes comfortable, as standard ProActive communication mechanism can be used for announcing actual URIs.

As already mentioned, the Data Spaces API can be run against variety of data transfer protocols that are provided by the infrastructure, or even against a standard ProActive-compliant communication protocols. This is achieved through pa-dataserver that is to be used wherever no supported data transfer protocol is available, but still the ProActive deployment can take place successfully. Hence, Data Spaces API can be used on almost every infrastructure.

The programmer can use ProActive's Data Spaces API to read application's input data, write and share temporary results and finally gather outputs in uniform manner among all these operations. Hence there are three types of data space abstraction: input, output and scratch. Data Spaces API provides way to access existing data spaces and files within these data spaces, it also gives the ability to define new data spaces. All operations performed through this API concern data spaces owned by one application.

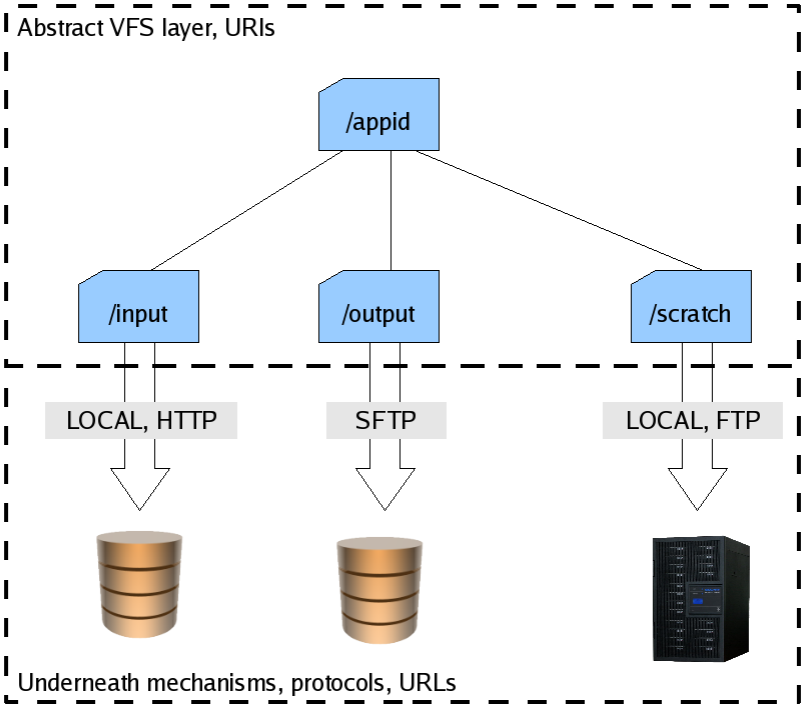


Figure 10.1. Abstract VFS layer used in Data Spaces (simplified). Each space is described by URI path that resolves to underneath physical protocols.

Every file in Data Spaces API is identified by an unique virtual URI. A given file can be accessed via several protocols at once if the data space was created using several physical protocols.

The URI can be shared between Active Objects even if they reside on different nodes. Files are organized in data stores called data spaces, grouped by their purpose and access. To access a file one can use designated data space to resolve file's name or directly resolve URI received from another Active Object.

There are distinguished three different space types: input space with files used as the application's input data, output space for storing the application's result files and scratch space for temporary files with the intermediate results of the computations. Spaces types differ in the access rights, and therefore input is only meant for read operations, output for read and write as well, whereas scratch's access mode depends on the caller. If a calling thread is an owner of a particular scratch, then it can perform both read and write operations. On the other hand, if a calling thread is not an owner (eg. has received URI from another Active Object), its access is limited to read only operations.

Input and output spaces are mostly defined by the application, hence their configuration is stored in the application-specific GCM-A Descriptor or in the application code itself. An application can have several inputs and outputs defined, as each input and output space has its unique (within a single application) name. There can be defined a default (single) input or output spaces, so using names is not always needed. For more information please refer to attached JavaDoc.

Scratch data space is most likely defined by the system administrator in the GCM-D Descriptor in case of the GCM deployment, or manually. Scratch data space is bounded to Node, and particular scratches within a single scratch data space are bounded to Active Objects that reside on this Node. Hence, there is only one scratch that Active Object owns.

	Space type	Description	Access
Application	*Input space*	Input for an application	RO
Application	*Output space*	Output for an application	RW
Nodes	*Scratch space*	Temporary storage of intermediate results	RW/RO*

Figure 10.2. Data spaces categories, purpose and access rights. (* for scratch space RW access only for owning AO)

10.2. Configuring Data Spaces

10.2.1. Data Spaces and GCM Deployment

The easiest and typical way to configure Data Spaces is using it together with GCM Deployment, which is described in Chapter 14, *ProActive Grid Component Model Deployment*.

When using Data Spaces with GCM Deployment, all necessary configuration actions are implemented inside GCM deployment mechanism. User and/or administrator just needs to specify configuration in GCM Application and GCM Deployment descriptor files, as usually with this kind of deployment. Once Data Spaces configuration is placed in GCM descriptors, user can access Data Spaces API from every node deployed through GCMApplication.

10.2.1.1. GCM Application Descriptor

In GCM Application Descriptor user defines a few application-wide settings:

- whether to **enable or disable Data Spaces** for the application,
- (optionally) **Naming Service settings**,
- (optionally) application's **input and output data spaces** definitions.

Let us have a look how example GCM Application Descriptor may look like:

```
...
<application>
<proactive>
  ...
  <virtualNode>
  ...
  </virtualNode>
  <data>
    <inputDefault>
      <remoteAccess url="sftp://storage.company.com/var/input_data/" />
    </inputDefault>

    <outputDefault>
```

```
        <remoteAccess url="sftp://bob@output_host.company.com/tmp/output/" />
        <location hostname="output_host.company.com" path="/tmp/output/" />
      </outputDefault>
      <output id="stats">
        <remoteAccess url=ftp://julia:password@another_host.company.com/home/julia/stats/" />
      </output>
    </data>
</proactive>
</application>
...
```

Example 10.1. GCMA descriptor fragment - Data Spaces configuration

Step by step - how configuration is specified:

- **<data>** tag.

  Presence of this tag determines whether Data Spaces will be enabled for the whole application. There are no attributes for this tag. All other Data Spaces related tags are included in this one.

- **<namingService>** tag (optional).

  Data Spaces needs so called Naming Service to function properly, which acts as a directory of data spaces. This tag is responsible for Naming Service related settings.

  For typical configuration it is appropriate to omit this tag, which stands for starting own Naming Service for deployed GCMApplication and closing it with the end of that application life time.

  In some configurations you may want to use already started Naming Service. In that case you may specify its URL within url attribute, in the following way:

  ```
  <namingService url="rmi://branch.inria.fr:1099/5933540059531990841/namingService" />
  ```

- **Input and output specification** tags (optional).

  These group of tags specify input and output data spaces for the application. Spaces has to be defined in a specific order: default input space **<inputDefault>**, named input spaces **<input>**, default output space **<outputDefault>** and named output spaces **<output>**. Each of them is optional.

  Every input or output data space is defined in almost the same way. Such a definition consists of the following tags/attributes:

  - <remoteAccess> tag with url attribute.

    This mandatory part of space definition sets URL that will be used for accessing that data space. URL may point to a file or directory. For output space file may not exist yet.

  - <location> tag with hostname and path attributes (optional).

    If this tag is present in space definition, data will be accessed locally on a host specified in hostname, under local path specified in path. This path should represent the same data as remoteAccess tag does.

  - id attribute (only for non-default, named spaces).

    Non-default, named input or output spaces have to be identified by unique id, which is set through that attribute.

  Example data space definition:

  ```
  <input id="precomputation">
    <remoteAccess url="sftp://bob@storage.com/var/precomputed/" />
    <location hostname="storage.com" path="/var/precomputed/" />
  </input>
  ```

10.2.1.2. GCM Deployment Descriptor

In GCM Deployment Descriptor grid administrator defines configuration of a scratch data space for each host configuration. It is done by putting settings inside host definition, like in the following example:

```
...
<host id="host" os="${os}" hostCapacity="2" vmCapacity="1">
   <homeDirectory base="root" relpath="${user.home}" />
   <scratch>
      <remoteAccess url="sftp://bob@#{hostname}/tmp/dataspaces" />
      <path base="root" relpath="/tmp/dataspaces" />
   </scratch>
</host>
...
```

Example 10.2. GCMD descriptor fragment - Data Spaces configuration

As you can see, all Data Spaces related configuration is contained within **<scratch>** tag.

If this tag is not present, there will be no scratch data space configured for Nodes with that host configuration, i.e. PADataSpaces.resolveScratchForAO() calls will fail on Node with that host configuration. If the tag is present, scratch data space will be configured on Nodes with that configuration. Definition of scratch configuration consists of two tags, whereas at least one of them have to be present:

- <remoteAccess> tag with url attribute.

  This part of scratch space definition is similar to input and output spaces configuration - it sets URL that will be used for accessing that data space. Any existing file that URL points to will be deleted and replaced with a base scratch space directory. Note that special metavariable #{hostname} may be used in URL template, as the same generic host configuration may be applied to more than one real hosts. #{hostname} is replaced with actual Node's hostname during deployment.

  Absence of <remoteAccess> tag have a special meaning in context of scratch configuration. If the tag is not present, ProActive provider server will be automatically started for a Node lifetime, exposing this part of a file system remotely through ProActive protocol, so you do not need to configure and run external protocol server.
- <path> tag with base and relpath attributes.

  Path definition part specifies location of scratch space on host's local file system tree. Path specified in relpath is relative to base, which may be one of following values: proactive, home or root (refer to Chapter 14, *ProActive Grid Component Model Deployment* for details).

  When <remoteAccess> tag is present, local path specification is just used for optimizing local data access. If <remoteAccess> is not present, this path is also used as a root of file system exposed through ProActive provider server. In any configuration, if both local path and access URL are provided, they should point to the same data.

10.2.2. Command-line tools

Data Spaces provide some command-line tools that you may find useful in some configurations. Note that they are NOT REQUIRED to be used in basic configurations.

10.2.2.1. namingService script

This shell script allows you to perform **operations on Naming Service**. In particular, it allows starting standalone Naming Service, which can be shared among several applications. To start Naming Service, use script with start command:

```
linux > ./namingService.sh start
```

```
Naming Service successfully started on: rmi://branch.inria.fr:1099/defaultNamingService
```

As you can see, you get information with URL of started Naming Service, that you can later use in GCM Application descriptor or for manual configuration, instead of starting separate Naming Service for each application. See Section 10.2.1.1, "GCM Application Descriptor" and Section 10.2.3, "Manual configuration" for configuration details.

You can also start Naming Service within your Java code using NamingServiceDeployer class.

namingService script can also be used for monitoring or low-level operations, through ls and add commands.

10.2.2.2. pa-dataserver script

Using this script you can **start ProActive provider server**, exposing specified local directory for a remote access. ProActive provider bases on ProActive protocol to create remote file system protocol. Performance of such a protocol may be worse than performance of dedicated protocols like FTP, but it can act as an easy to setup protocol, without lot of configuration nor requirements. To start ProActive provider server, simply use script with start command:

```
linux > ./pa-dataserver.sh start /home/bob/input_data/
70833@branch - [INFO    communication.rmi] Created a new registry on port 1099
ProActive dataserver successfully started.
VFS URL of this provider: paprmi://branch.inria.fr:1099/defaultFileSystemServer?proactive_vfs_provider_path=/
```

Displayed URL of started server can be used as a remote access URL of data spaces in GCM Application descriptor, PADataSpaces API or for manual configuration, to easily set up remote access for some data. See Section 10.2.1.1, "GCM Application Descriptor" and Section 10.2.3, "Manual configuration" for configuration details and Example 10.4, "Dealing with inputs more dynamically — adding named input, reading all defined inputs" or javadoc for API details.

You can also start ProActive provider server within your Java code using FileSystemServerDeployer class.

ProActive provider server uses two ProActive properties (see Section 11.4, "ProActive properties" for details), to adjust **streams autoclosing** feature. This mechanism is meant to close streams those were not being used for a long time, and automatically reopen them if necessary. It can be used to save resources or avoid orphaned resources. Following properties can be set:

- proactive.vfsprovider.server.stream_autoclose_checking_millis - indicating how often an auto closing mechanism is started to collect and close all unused streams open (default: 30 000).
- proactive.vfsprovider.server.stream_open_maximum_period_millis - indicating a period after that a stream is perceived as unused and therefore can be closed by auto closing mechanism. (default: 60 000).

ProActive system properties (see Section 11.4, "ProActive properties" for details) can also be set using command line (-Dproperty=value). For instance, the port can be configured using -Dproactive.rmi.port=1099 given that RMI is used.

10.2.3. Manual configuration

In some environments it may be useful to access Data Spaces configuration API manually. Note that most of the users do NOT NEED to know anything about this API, but it is exposed to give more flexibility for potential usages. When you need to configure Data Spaces manually, you need to be aware of a few assumptions that Data Spaces bases on, which are normally fulfilled by GCM deployment implementation.

First of all, Data Spaces uses the concept of **application** operating on data spaces. Every application is uniquely identified by **application id**. Set of input, output and scratch data spaces is always associated with some concrete application and typically application accesses only its own data spaces.

Secondly, Data Spaces assumes that each Node is used at most by one application at any given time, i.e. **Node sharing is not allowed** between applications using Data Spaces.

Configuration of Data Spaces can be divided into two units - **Naming Service-level** which is common to the whole application and set up once, and **Node-level** configuration which is individually applied for each Node. Next two sections discuss these two

configuration units in more details, while you can find complete manual configuration example in ManulConfigurationExample class from org.objectweb.proactive.examples.dataspaces.manualconfig package.

10.2.3.1. Naming Service-level configuration

Data Spaces needs a running service called Naming Service, which acts as a directory of data spaces. One Naming Service may be used to serve more than one application. Each application needs to be explicitly registered and unregistered there. Therefore, unique application identifier is required.

Naming Service can be **started** from Java code using NamingServiceDeployer class or using Section 10.2.2, “Command-line tools”:

```
// start Naming Service
final NamingServiceDeployer namingServiceDeployer = new NamingServiceDeployer();
final String namingServiceURL = namingServiceDeployer.getNamingServiceURL();
```

Once Naming Service is started and can be accessed, application that will use Data Spaces needs to **register** there with optional set of initial input and output spaces:

```
// need to guarantee uniqueness of application id somehow
final long applicationId = 1234431;
// @snippet-bDataSpacesManualConfig_RegisteringApp
// create set of predefined inputs and outputs - here: default input accessed via FILE and HTTP
final InputOutputSpaceConfiguration inSpaceConf = InputOutputSpaceConfiguration
      .createInputSpaceConfiguration(Arrays.asList("file:///user/share/ftp/rfc/rfc2616.txt",
            "http://www.faqs.org/ftp/rfc/rfc2616.txt"), null, null,
            PADataSpaces.DEFAULT_IN_OUT_NAME);
final SpaceInstanceInfo inSpaceInfo = new SpaceInstanceInfo(applicationId, inSpaceConf);
final Set<SpaceInstanceInfo> predefinedSpaces = Collections.singleton(inSpaceInfo);

// access (possibly remote) Naming Service
final NamingService namingService = NamingService.createNamingServiceStub(namingServiceURL);
// register application
namingService.registerApplication(applicationId, predefinedSpaces);
```

When application terminates, it should be deconfigured by **unregistering** itself from Naming Service:

```
namingService.unregisterApplication(applicationId);
```

If Naming Service was started through NamingServiceDeployer and is not used anymore, it can be **stopped**:

```
namingServiceDeployer.terminate();
```

10.2.3.2. Node-level configuration

Configuration of Node is performed through DataSpacesNodes class and consists of two steps. Every Node-level configuration needs to be performed locally on a Node that is being configured, it cannot be done remotely without any agent Active Object or other mechanism.

The first step is to **initialize Node for Data Spaces** in general, optionally providing base of scratch space configuration that will be later used by applications on this Node. This step is application agnostic, and it is usually performed only once for a Node.

```
// prepare base scratch configuration - here: using tmp dir,
// with null url - exposed through automatically started ProActive provider server
final String tmpPath = System.getProperty("java.io.tmpdir") + File.separator + "scratch";
// configure node for Data Spaces
final BaseScratchSpaceConfiguration scratchConf = new BaseScratchSpaceConfiguration((String) null,
   tmpPath);
```

```
DataSpacesNodes.configureNode(node, scratchConf);
```

The second step is to **configure Node for a particular application**, specifying unique application id and Naming Service URL.

```
// configure node for application
DataSpacesNodes.configureApplication(node, applicationId, namingService);
```

When Node is configured for Data Spaces this way, Active Objects bodies (or non-Active Objects half bodies) can safely use PADataSpaces API.

Once Node has been configured for some application, it can be explicitly **deconfigured for that application** - when application is not using this Node anymore:

```
DataSpacesNodes.tryCloseNodeApplicationConfig(node);
```

...or it can be **reconfigured for another application** - when another application is starting to use it in place of previous one:

```
DataSpacesNodes.configureApplication(node, anotherApplicationId, namingServiceURL);
```

If Node will not be used anymore for any application (e.g. it is going to be destroyed), its Data Spaces configuration can be completely **closed**:

```
DataSpacesNodes.closeNodeConfig(node);
```

10.3. Using Data Spaces API

10.3.1. Code snippets and explanation

Below you can find a few simple examples that show how to deal with basic tasks using Data Spaces API. Those examples remain just ideas of how Data Spaces API can be used, we encourage you to invent your own tricky-application.

10.3.1.1. Reading from default input, writing to named output

The first code snippet shows how to read input of an application and to write the final results (here: some statistics). For reading input data, a default input space is used, as there is only one input store used for this example. Statistics are stored in the named output data space according to STATS_DATA_SPACE constant, that could be a hard coded identifier or even an application parameter.

Methods from the Data Spaces API used in this snippet are: PADataSpaces.resolveDefaultInput and PADataSpaces.resolveOutput that resolve default input and named output respectively, returning instance of DataSpacesFileObject class. This class is a part of Data Spaces API and represents a file (ordinal file or directory) from any data space and allows to read or write file's content through standard Java APIs streams along with another file system operations. For a convenience, a file name can be specified in the parameters list of above mentioned methods.

```
public void loadInput() {
   DataSpacesFileObject inFile = PADataSpaces.resolveDefaultInput("data.txt");
   InputStream is = inputFile.getContent().getInputStream();
   // read from stream
   is.close();
   inputFile.close();
}

private final static String STATS_DATA_SPACE = "stats";

public void saveStats() {
   DataSpacesFileObject statsFile = PADataSpaces.resolveOutput(STATS_DATA_SPACE, id + ".txt");
   OutputStream os = statsFile.getContent().getOutputStream();
```

```
    // ...
}
```

Example 10.3. Reading from default input and writing to named output

10.3.1.2. Adding named inputs dynamically, reading all defined inputs

The next snippet could be part of an application composed of a GUI that allows user to define inputs dynamically in the run time, for the further processing. Inputs are added (registered for the application) on user's demand by addInputClicked event handler (made up for this example), that calls Data Spaces API routine PADataSpaces.addInput and informs other processing objects (Active Objects) about the new input. The processing can be done as follows. Using the PADataSpaces.resolveAllKnownInputs method one can iterate over all inputs registered in the moment of resolving. Hence, for each such an input, a processing (reading particular files) can take place.

```
public void addInputClicked() {
    PADataSpaces.addInput(gui.getName(), gui.getURL()), gui.getPath());
    informOthers();
}

public void processAllInputs() {
    for (Entry<String, DataSpacesFileObject> input : PADataSpaces.resolveAllKnownInputs()) {
        String name = input.getKey();
        DataSpacesFileObject inputDir = input.getValue();

        if (isAlreadyProcessed(name))
            continue;
        DataSpacesFileObject children[] = inputDir.getChildren();
        // process each file in each input...
    }
}
```

Example 10.4. Dealing with inputs more dynamically — adding named input, reading all defined inputs

10.3.1.3. Sharing scratch URI between two Active Objects

This snippet presents how two Active Objects can share data stored in ones scratch. The scenario is as follows. One Active Object resolves a file from its scratch with PADataSpaces.resolveScratchForAO Data Spaces APIs method. Once the file is resolved (represented by an instance of DataSpacesFileObject class), the intermediate results can be written into its content. As each file within data spaces is identified by its URI, such a URI can be obtained with getURI method of DataSpacesFileObject class and passed to another Active Object as a parameter of exemplary readMyFile method. This method (in a different Active Object or even node!) can finally resolve URI through PADataSpaces.resolveFile method call and further process the file's content.

```
public void computeAndStore() {
    // compute...
    final DataSpacesFileObject scratchFile = PADataSpaces.resolveScratchForAO("data.txt");
    // write intermediate results there...
    scratchFile.close();
    anotherAO.readMyFile(scratchFile.getURI());
}
public void readMyFile(String uri) {
    DataSpacesFileObject file = PADataSpaces.resolveFile(uri);
    // process this file...
}
```

Example 10.5. Writing into ones scratch, that is later read by another AO

10.3.2. Complete example

In this section a complete example of processing data with ProActive and Data Spaces API is presented. Source code can be found in org.objectweb.proactive.examples.dataspaces.hello package, whereas xml descriptors in the examples/dataspaces/hello/ sub directory of your ProActive installation.

The goal of the processing is simple: count the number of lines in a text documents and store the results in a single text file. The computations are performed as follows. Every input document is mapped onto one Active Object so that inputs can be read and processed (lines counted) in parallel. Number of lines in each document (so called intermediate results) are stored in AO's scratches, so one can finally read them and gather into one file placed in the output data space. All these operations are simply done using the Data Spaces API.

10.3.2.1. GCM Deployment Descriptor

The GCM-D is very simple, as the example is to be run on every infrastructure. In this case, a **<scratch>** tag is used to enable data spaces on nodes described by the id="host" host id. A **path** element defines a location of a scratch data space, as a sub directory of the system's temporary directory. As there is no **<remoteAccess>** element, the scratch data space is to be exposed through ProActive dataserver that is started automatically.

```
<infrastructure>

  <hosts>
    <host id="host" os="${os}" hostCapacity="2" vmCapacity="1">
      <homeDirectory base="root" relpath="${user.home}" />
      <scratch>
        <path base="root" relpath="${java.io.tmpdir}/dataspaces" />
        <!--
          Use this if you prefer SFTP rather than ProActiveProvider file access:
          <remoteAccess url="sftp://${user.name}@#{hostname}${java.io.tmpdir}/dataspaces"/>
        -->
      </scratch>
    </host>
  </hosts>

  <groups>
    <sshGroup id="remoteThroughSSH" hostList="${HOST}" />
  </groups>

</infrastructure>
```

Example 10.6. Part of the GCM-D Descriptor: helloDeploymentRemote.xml

10.3.2.2. GCM Application Descriptor

GCM-A part contains application related information regarding data spaces configuration like inputs and outputs definitions, as well as contract variables used in the example code. The latest is defined as follows:

```
<programVariable name="OUTPUT_HOSTNAME" />
```

and represents name of a host that the output data is to be stored on.

The data spaces configuration part needs somewhat more explanation. It is placed right after Virtual Node definition, as a last child of the **<proactive>** element.

```
  </virtualNode>
  <data>
```

```
    <!--  input data spaces - HTTP resources to process -->
    <input id="wiki_proactive">
      <remoteAccess url="http://en.wikipedia.org/wiki/ProActive" />
    </input>
    <input id="wiki_grid_computing">
      <remoteAccess url="http://en.wikipedia.org/wiki/Grid_computing" />
    </input>
    <!--  default output data space for results  -->
    <outputDefault>
      <remoteAccess
        url="sftp://${user.name}@${OUTPUT_HOSTNAME}${user.home}/tmp/output/" />
      <location hostname="${OUTPUT_HOSTNAME}" path="${user.home}/tmp/output/" />
    </outputDefault>
  </data>
</proactive>
```

Example 10.7. Part of the GCM-A Descriptor: helloApplication.xml

Descriptor defines two named input spaces that serve text resources for our example , a document's lines counter. In this particular case, two resources are accessed by the HTTP protocol, as they are two Wikipedia articles on ProActive library[1] and on grid computing[2]. Those inputs can be later resolved by an application using their names (identifiers), here: wiki_proactive and wiki_grid_computing respectively.

The output space is placed on the OUTPUT_HOSTNAME host. Output space is a directory placed in the user's home directory and can be accessed remotely by the SFTP protocol or locally, as an explicit location is given.

10.3.2.3. The Java code

The HelloExample.java file contains code responsible for deployment and computation scenario in general. At the beginning, the variable contract is fulfilled, passing the deployer's host name into descriptor. Therefore the output data space is now fully defined , placed on the deployer's host.

```
private void setupVariables() {
  vContract = new VariableContractImpl();
  // this way of getting hostname is not the best solution, but it makes
  // local execution of example possible without using protocols like SFTP
  vContract.setVariableFromProgram(VAR_OUTPUT_HOSTNAME, Utils.getHostname(),
      VariableContractType.ProgramVariable);
}
```

Example 10.8. HelloExample.java: setting up variable from the contract

The GCM deployment used in this example is a common way to deploy your application, as you can see, no additional work is required to enable the Data Spaces API.

```
private void startGCM(String descriptorPath) throws ProActiveException {
  gcmApplication = PAGCMDeployment.loadApplicationDescriptor(new File(descriptorPath), vContract);
  gcmApplication.startDeployment();

  final GCMVirtualNode vnode = gcmApplication.getVirtualNode(VIRTUAL_NODE_NAME);
  vnode.waitReady();
```

[1] http://en.wikipedia.org/wiki/ProActive
[2] http://en.wikipedia.org/wiki/Grid_computing

```
    // grab nodes here
    nodesDeployed = vnode.getCurrentNodes();
    logger.info("Nodes started: " + nodesDeployed.size() + " nodes deployed");
}
```

Example 10.9. HelloExample.java: starting the GCM deployment

All routines related to processing scenario are gathered into one exampleUsage method (Example 10.10, “HelloExample.java: computation scenario used in the example”). At the very beginning, nodes grabbed previously during the deployment phase are now used for placing two Active Objects instantiated from the ExampleProcessing class. Both are later called to perform computePartials method, that obtains as a parameter the name of an input space to process. Input space names are defined as follows:

```
public static final String INPUT_RESOURCE1_NAME = "wiki_proactive";

public static final String INPUT_RESOURCE2_NAME = "wiki_grid_computing";
```

and thereby reflect the GCM-A input spaces definition. Partial results (here: number of lines in each document) are stored in text files in the AO's scratches, that URIs appear as a computePartials method's return value. Collection of those URI's is later passed to gatherPartials method called on one of instantiated Active Objects.

```
private void exampleUsage() throws ActiveObjectCreationException, NodeException, DataSpacesException {
    checkEnoughRemoteNodesOrDie(2);
    final Node nodeA = nodesDeployed.get(0);
    final Node nodeB = nodesDeployed.get(1);

    final ExampleProcessing processingA = PAActiveObject.newActive(ExampleProcessing.class, null, nodeA);
    final ExampleProcessing processingB = PAActiveObject.newActive(ExampleProcessing.class, null, nodeB);
    final Collection<StringWrapper> partialResults = new ArrayList<StringWrapper>();
    try {
        partialResults.add(processingA.computePartials(INPUT_RESOURCE1_NAME));
        partialResults.add(processingB.computePartials(INPUT_RESOURCE2_NAME));
    } catch (IOException x) {
        logger.error("Could not store partial results", x);
        return;
    }

    try {
        processingB.gatherPartials(partialResults);
    } catch (IOException x) {
        logger.error("Could not write final results file", x);
    }
}
```

Example 10.10. HelloExample.java: computation scenario used in the example

Finally, let's have a look at routines that are performed by Active Objects instantiated from the ExampleProcessing class (Example 10.11, “ExampleProcessing.java: routines performed by Active Objects (1)” , refer to comments for the routines' explanation. There are two main processing methods computePartials and gatherPartials shown, along with the writeIntoScratchFile helper method.

```
public String writeIntoScratchFile(String fileName, String content) throws NotConfiguredException,
```

```
        IOException, ConfigurationException {
    DataSpacesFileObject file = null;
    OutputStreamWriter writer = null;

    try {
        // resolve scratch for this AO, and get Data Spaces file representation of fileName;
        // later, be sure that the file was created and open an output stream writer on this file;
        file = PADataSpaces.resolveScratchForAO(fileName);
        file.createFile();
        writer = getWriter(file);
        // finally, write the content and return files URI, valid for every AO
        writer.write(content);

        return file.getVirtualURI();
    } catch (IOException e) {
        logger.error("Exception while IO operation", e);
        throw e;
    } finally {
        closeResource(writer);
        closeResource(file);
    }
}
```

Example 10.11. ExampleProcessing.java: routines performed by Active Objects (1)

```
public StringWrapper computePartials(String inputName) throws SpaceNotFoundException,
        NotConfiguredException, IOException, ConfigurationException {

    logger.info("Processing input " + inputName);
    DataSpacesFileObject inputFile = null;
    BufferedReader reader = null;
    int lines = 0;

    try {
        // resolve a named input that's name was passed as a method's parameter
        // as input represents file that's content is to be processed, open a reader
        inputFile = PADataSpaces.resolveInput(inputName);
        reader = getReader(inputFile);

        // count lines here..
        while (reader.readLine() != null)
            lines++;

        StringBuffer sb = new StringBuffer();
        sb.append(inputName).append(": ").append(lines).append('\n');

        // store the partial result in a file within AO's scratch
        String fileUri = writeIntoScratchFile(PARTIAL_RESULTS_FILENAME, sb.toString());
        logger.info("partial results written: " + sb.toString());

        // finally return file's URI
        return new StringWrapper(fileUri);
    } catch (IOException e) {
```

```
            logger.error("Exception while IO operation", e);
            throw e;
        } finally {
            closeResource(reader);
            closeResource(inputFile);
        }
    }
```

Example 10.12. ExampleProcessing.java: routines performed by Active Objects (2)

```
    public void gatherPartials(Iterable<StringWrapper> partialResults) throws MalformedURIException,
            DataSpacesException, IOException {
        logger.info("Gathering and aggregating partial results");

        final List<String> results = new ArrayList<String>();

        // for every URI that was passed...
        for (StringWrapper uriWrapped : partialResults) {
            DataSpacesFileObject partialResultsFile = null;
            BufferedReader reader = null;
            try {
                // ... resolve file pointed by that URI and open a reader, as it contains partial result
                partialResultsFile = PADataSpaces.resolveFile(uriWrapped.getStringValue());
                reader = getReader(partialResultsFile);

                // ... and gather partial results in a list
                results.add(reader.readLine());
            } catch (IOException x) {
                logger.error("Reading one's partial result file failed, trying to continue", x);
            } finally {
                closeResource(reader);
                closeResource(partialResultsFile);
            }
        }

        DataSpacesFileObject outputFile = null;
        OutputStreamWriter writer = null;
        try {
            // resolve a file from the default output space (as such is been defined in the GCM-A);
            // be sure that the file exists and write gathered results using the output stream writer
            outputFile = PADataSpaces.resolveDefaultOutput(FINAL_RESULTS_FILENAME);
            outputFile.createFile();
            writer = getWriter(outputFile);

            for (String line : results)
                if (line != null) {
                    writer.write(line);
                    writer.write('\n');
                }
            logger.info("Results gathered, partial results number: " + results.size());
        } catch (IOException e) {
            logger.error("Exception while IO operation", e);
            throw e;
```

```
    } finally {
        closeResource(writer);
        closeResource(outputFile);
    }
}
```

Example 10.13. ExampleProcessing.java: routines performed by Active Objects (3)

10.3.2.4. Example's output

In this section an output of above mentioned example is presented, both: a simplified logger output and the output file's content.

```
# simplified logger's output (info messages from the example of each thread):
[INFO   proactive.examples] Nodes started: 2 nodes deployed
[INFO   proactive.examples] Processing input wiki_proactive
[INFO   proactive.examples] partial results written: wiki_proactive: 514
[INFO   proactive.examples] Processing input wiki_grid_computing
[INFO   proactive.examples] partial results written: wiki_grid_computing: 732
[INFO   proactive.examples] Gathering and aggregating partial results
[INFO   proactive.examples] Results gathered, partial results number: 2
[INFO   proactive.examples] Application stopped

# final results of the processing (order of lines may differ after each run):
$ cat tmp/output/final_results.txt
wiki_proactive: 514
wiki_grid_computing: 732
```

Part III. ProActive Configuration

Table of Contents

Chapter 11. ProActive Basic Configuration

11.1. 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.

11.2. 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 or RuntimeFactory class (static block in all that classes).

For instance, calling **PAActiveObject.newActive or NodeFactory.getNode** will load 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'/>
```

11.3. Where to access this file?

11.3.1. ProActive Default Configuration file

There is a default file with default ProActive options located under ProActive/src/org/objectweb/proactive/core/config/ProActiveConfiguration.xml. 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).

11.3.2. User-defined ProActive Configuration file

It is possible for a user to override the default ProActive configuration file by setting properties in a file located at the following path:

```
for unix users: $HOME/.proactive/ProActiveConfiguration.xml
for windows users: $HOME\.proactive\ProActiveConfiguration.xml
```

11.3.3. Alternate User Configuration file

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

```
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 programmatically 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.

The specific tag <**userProperties**> is provided in application descriptors to notify the remote JVMs which configuration file to load once created:

```
<application>
  <proactive base="root" relpath="${proactive.home}">
    <configuration>
      <userProperties base="proactive" relpath="MyProActiveConfiguration.xml"/>
    </configuration>
  </proactive>
</application>
```

To know more about GCM deployment, please refer to Chapter 14, *ProActive Grid Component Model Deployment*.

11.4. ProActive properties

All the properties described here after are defined in the org.objectweb.proactive.core.config.PAProperties class. To have an exhaustive list of ProActive properties, please refer to this class.

11.4.1. Required

- **proactive.communication.protocol**: Represents the communication protocol, i.e. the protocol used to export objects on remote JVMs. At this stage, several protocols are supported: **RMI(rmi), HTTP(http), IBIS/RMI(ibis), SSH tunneling for RMI/HTTP(rmissh), ProActive Message Routing(pamr)**. It means that once the JVM starts, nodes and 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**: Boolean property indicating whether to validate XML files against the provided schema.Default is **true**
- **proactive.future.ac**: Boolean value saying whether to activate Automatic Continuations (see Section 2.8, "Automatic Continuation in ProActive"). Default is **true**

Note

If not specified, those properties are set programmatically with the default value.

11.4.2. Fault-tolerance properties

Note that those properties should not be altered if the programmer uses deployment descriptor files. See Chapter 32. Fault-Tolerance[1]. and more specifically Chapter 32. Fault-Tolerance[2]. for more details.

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/faultTolerance.html

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/faultTolerance.html#faultTolerance_html_configuration

- **proactive.ft**: Boolean indicating whether to enable the fault-tolerance and if so, a set of servers has to be defined with the following properties. Default value is **false**.
- **proactive.ft.server.checkpoint**: URL of the checkpoint server.
- **proactive.ft.server.location**: URL of the location server.
- **proactive.ft.server.recovery**: URL of the recovery process.
- **proactive.ft.server.resource**: URL of the resource server.
- **proactive.ft.server.global**: URL of the global server. If this property is set, all others **proactive.fr.server.*** are ignored.
- **proactive.ft.ttc**: Value of the Time To Checkpoint counter (in seconds). The default value is **30 sec**.

11.4.3. rmi ssh properties

The following properties are specific to the rmissh protocol .

- **proactive.communication.rmissh.port**: Port number on which all the ssh daemons to which this JVM has to connect to are expected to listen. The default value is **22**.
- **proactive.communication.rmissh.username**: Username which will be used during authentication with all the ssh daemons to which this JVM will need to connect to. The default is the **user.name** java property.
- **proactive.communication.rmissh.known_hosts**: 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. The default is **System.getProperty('user.home') + '/.ssh/known_hosts'**
- **proactive.communication.rmissh.key_directory**: 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 has to be suffixed by '.pub'. Private keys are ignored if their associated public key is not present. The default is **System.getProperty('user.home') + '/.ssh/'**.
- **proactive.communication.rmissh.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 The default is **no** meaning these direct-connection will not be attempted. 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 front-end 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 front-end.
- **proactive.communication.rmissh.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. The default value is **2000 ms**.
- **proactive.communication.rmissh.gc_idletime**: This property identifies the maximum idle time before a SSH tunnel or a connection is garbage collected.
- **proactive.communication.rmissh.gc_period**: This property specifies how long the tunnel garbage collector will wait before destroying an unused tunnel. If a tunnel is older than this value, it is automatically destroyed. The default value is **10000 ms**.

11.4.4. 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 uses 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 transparently incremented).
- **proactive.useIPaddress**: If set to **true**, IP addresses will be used instead of machines names. This property is particularly useful 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 useful 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.

Note

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

11.5. Configuration file example

A configuration file could have following structure:

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

Example 11.1. A configuration file example

Note

In order to have ProActive parse correctly the document, the following are mandatory:

- the **ProActiveUserProperties** tag,
- the **properties** tag,
- and the model: <prop **key**='somekey' **value**='somevalue'/>

11.6. Log4j configuration

ProActive Programming uses log4j as logging framework. This section presents various log4j setups for distributed applications. It also introduces the ProActive log4j appender. It aims at allowing easy centralization of all the logging events created by a distributed application. It relies on ProActive communications to send logging events from computational resources to a collector.

Log4j basics are not explained here. If you want to learn more about log4j, please refer to the log4j introduction[3].

11.6.1. ProActive Appender

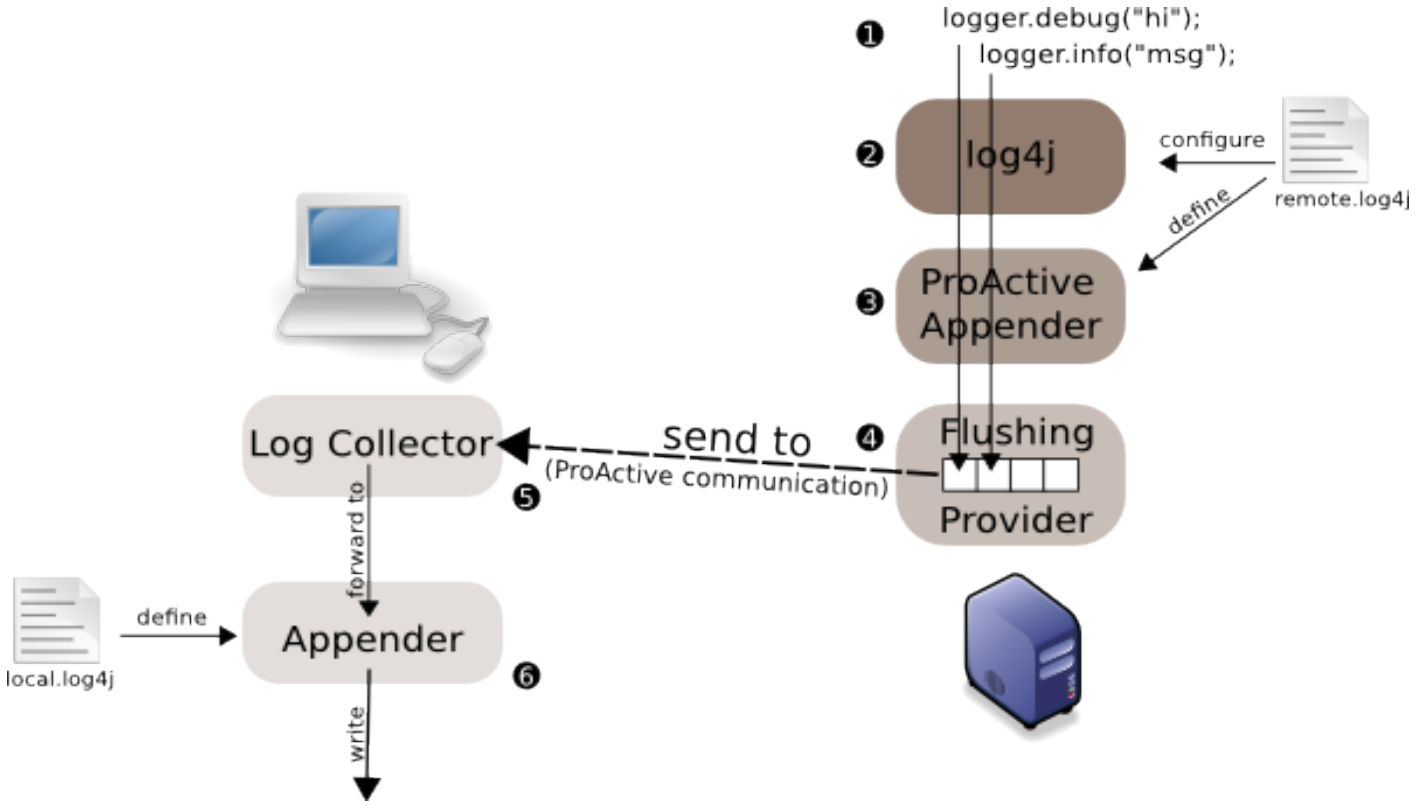
Before ProActive Programming 4.1.0, a default log4j configuration file was shipped with the ProActive Programming jar file. This configuration file defined a ConsoleAppender to log all the logging events on the standard output with a custom layout. With this configuration, it was not easy to gather all the logging events. Even if the GCM Deployment monitors the standard output of the forked processes, job scheduler like LSF or PBS redirect the standard output to a file. The user had to collect the log by itself.

In addition it was not easy to write logging events into a file. log4j instances are not synchronized. Using the same **FileAppender** in two JVMs leads to corrupted log files since several processes write in the same file concurrently.

ProActive Programming 4.1.0 introduced the ProActiveAppender. It allows the deployed ProActive runtimes to forward their logging events to their parent runtime through ProActive communications. It is now easy to centralize all the logging events created by your application at a unique location. Contrary to other distributed appender shipped with log4j, the ProActiveAppender is ready to use out-of-box and well integrated in the GCM Deployment framework. You don't have to write your own event collector or to setup an heavy weight infrastructure. In addition, the local log4j configuration of the log collector is used to store the local and remote logging events.

[3] http://logging.apache.org/log4j/1.2/manual.html

The following schema describes how the ProActiveAppender works:



Overview of the ProActiveAppender

Description of the different entities involved in distributed logging

local.log4j	A log4j configuration file. It defines the behavior of log4j on the machine in charge of logging event centralization.
remote.log4j	A log4j configuration file. It defines the behavior of log4j on the remote machines. Must define and use a ProActiveAppender.
log collector	Receives all the remote logging events and forwards them to the local log4j appenders. Defined by the local.log4j configuration file. Exported as a remote object.
ProActiveAppender	A log4j appender. It forwards logging events to a remote log collector through ProActive communications according to a flushing policy.
Flushing Provider	Defines how and when the logging events are sent to the remote log collector. Since different applications can have different requirements (example: should the logging events be buffered or dropped when the log collector is not available), it is possible to write and use a custom provider.

To explain how the ProActiveAppender works and to learn how to configure it, we will follow the path of a logging events.

A logging event is created when user code or ProActive invokes the Logger.info() or a similar methods (bullet 1). Once the logging event is created, log4j forwards it to a set of log4j appenders according to the current log4j configuration (bullet 2). To enable remote logging, a ProActiveAppender must be defined and used in the remote.log4j configuration file.

To forward all the logging events with a priority equals or higher than INFO, the following log4j configuration file can be used:

```
log4j.rootLogger=INFO,REMOTE

log4j.appender.REMOTE=org.objectweb.proactive.core.util.log.remote.ProActiveAppender
log4j.appender.REMOTE.layout=org.apache.log4j.PatternLayout
log4j.appender.REMOTE.layout.ConversionPattern=%X{shortid@hostname} - [%p %20.20c{2}] %m%n
```

Of course, it is possible to define several appenders (remote and local for example) or to use the remote appender of only a given set of loggers.

When instantiated by log4j, the ProActiveAppender expects to find the URI of the log collector in the proactive.log4j.collector java property. It also loads a flushing provider according to the proactive.log4j.appender.provider java property. If the property is not set, the first provider found is used.

ProActiveAppender retries the remote log collector, setup the flushing provider and forwards the logging events to the flushing provider (bullet 3). It also buffers all the logging events until the log collector is available. If it takes more than 30 seconds, logging events are printed on the standard output.

The flushing provider sends the logging events to the log collector via a ProActive communication (bullet 4). It defines its owns buffering and flushing policy. By default the org.objectweb.proactive.core.util.log.remote.ThrottlingProvider is used. It defines a simple but flexible and robust flushing policy. Logging events are buffered until one of the following conditions is met:

- The number of buffered messages has reached a threshold
- The buffer has not been flushed since a given period

The size of the buffer can be configured. When the buffer is full, callers are blocked until the logging events have been flushed. This behavior can slow down the application, but ensures that no logging events is lost. On commodity hardware, this provider can send almost 90000 logging events per seconds (two runtimes on the Intel Q9400).

When the log collector receives a set of logging events, it forwards each logging events to the local Logger which invoke the appenders defined by local.log4j (bullet 5). Appenders then write the logging events somewhere (file, standard output etc.).

11.6.2. ProActive Appender and GCM Deployment

This section presents how the GCM Deployment framework integrates the ProActiveAppender.

When the GCM Deployment framework starts a ProActive runtime, if no log4jProperties tag is defined in the GCM Application descriptor file, then the distributed-log4j is used as log4j configuration file. It sends all the logging events with a level equals or higher than INFO to a log collector located on the machine which has loaded the GCM Application descriptor. The logging events are handled by the appenders defined in the local.log4j file. Logging events filtering is done on the client side, not on the server side (i.e. all the logging events received by the log collector are forwarded to the appenders). To enable the DEBUG level on deployed machines, you have to write a remote.log4j configuration file and set the log4jProperties tag in the GCM Application descriptor file.

The following example of remote.log4j file enables the debug level for the application and the ProActive master worker framework:

```
log4j.rootLogger=INFO,REMOTE
log4j.logger.org.mortbay = WARN

log4j.logger.proactive.masterworker=DEBUG,REMOTE
log4j.logger.myapplication=DEBUG,REMOTE

log4j.appender.REMOTE=org.objectweb.proactive.core.util.log.remote.ProActiveAppender
log4j.appender.REMOTE.layout=org.apache.log4j.PatternLayout
log4j.appender.REMOTE.layout.ConversionPattern=%X{shortid@hostname} - [%p %20.20c{2}] %m%n
```

11.6.3. Other appenders

The ProActiveAppender is well suited for development and most production needs. It is easy to setup and well integrated to the ProActive GCM Deployment. But if your application generates a lot of logging events, or if you do not want to rely on ProActive to transport the logging events, you can use some other appenders provided with log4j. Useful appenders for distributed application are quickly described in this section.

The **JDBCAppender** sends logging events into a relational database via JDBC. The version shipped with log4j 1.2 is known to be broken and have some severe limitations. Alternative implementations of this appender exist.

The **SocketAppender** sends logging events to a remote log server. It is very similar to the ProActiveAppender but uses plain TCP connections. Main differences are:

- Remote logging uses TCP protocol
- If the remote log server is down, logging events are dropped.
- The SocketAppender does not flush buffered logging events on JVM exit. They are lost.

The **SocketHubAppender** sends logging events to several remote log server. Both only define the client side of distributed logging. You have to write your own SocketNode to handle the received logging events.

The **JMSAppender** publishes logging events to a JMS topic.

Chapter 12. Network configuration

ProActive offers several communication protocols. It is up to the user to decide which protocol to use according to its needs. Each protocol provides different features: speed, security, fast error detection, firewall or NAT friendliness, etc. None of the protocols can offer all these features, so they have to be chosen carefully. To take benefit from each communication protocol, a ProActive runtime can use several communication protocols concurrently.

12.1. Available communication protocols

This section enumerates all the available communication protocols and describes their benefits and drawbacks. Their configuration and options are also explained.

12.1.1. ProActive Network Protocol (PNP, pnp://)

12.1.1.1. Description

PNP is the general purpose communication protocol. Its performances are quite similar to RMI, but it is much more robust and network friendly: only one TCP port for each runtime, no shared registry, fast network failure discovery, better scalability.

A runtime using PNP binds to a given TCP port at startup. All incoming communications use this TCP ports. So deploying an application with PNP requires to open one and only one incoming TCP port. Two runtimes cannot share the same TCP port.

12.1.1.2. Configuration

To use PNP **proactive.communication.protocol** must be set to **pnp** . PNP behavior can be customized by setting these properties:

- **proactive.pnp.port** The TCP port to bind to. If not set PNP uses a random free port. If the specified TCP port is already used PNP will not start and an error message is displayed.
- **proactive.pnp.default_heartbeat** PNP uses heartbeat messages to monitor TCP socket and discover network failures. This value determines how long PNP will wait before the connection is considered broken. Heartbeat messages are usually sent every **default_heartbeat/2** ms. This value is in milliseconds. This value is a trade-off between fast error discovery and network overhead. The default value is 9000 ms. Setting this value to 0 disable the heartbeat mechanism and client will not be advertised of network failure before the TCP timeout (which can be really long).
- **proactive.pnp.idle_timeout** PNP channels are closed when unused to free system resources. Establishing a TCP connection is costly (at least 3 RTT) so PNP connections are not closed immediately but after a grace time. By default the grace time is 60 000 ms. Setting this value to 0 disable the autoclosing mechanism, connections are kept open forever.

12.1.2. ProActive Network Protocol over SSL (PNPS, pnps://)

12.1.2.1. Description

PNPS is the PNP protocol wrapped inside an SSL tunnel. It provides the same features as PNP plus ciphering and optionally authentication. Using SSL implies some CPU overhead and PNPS is slower than PNP.

12.1.2.2. Use cases

There are two uses for PNPS. The first one is to secure communication with ciphering when the messages can be sent over an untrusted network. Please note that PNPS in ciphering mode only prevent messages from being sniffed on the network. Any malicious user able to establish a TCP connection to the runtime can connect to it and execute any arbitrary code. In ciphering only mode the runtime must be configured as follows:

```
-Dproactive.communication.protocol=pnps [-Dproactive.pnps.keystore=file:///path/to/keystore.p12]
```

The second use case is ciphering and authentication. In addition to message ciphering this mode authenticate both the client and the server. To establish a PNPS connection to a runtime, both the client and the server must send a trusted certificate. This mode allows to secure a ProActive runtime from a malicious user and prevent sniffing. In authentication and ciphering mode the runtime must be configured as follows:

```
-Dproactive.communication.protocol=pnps -Dproactive.pnps.authenticate=true -Dproactive.pnps.keystore=file:///path/to/keystore.p12
```

12.1.2.3. Configuration

PNPS support the same options as PNP (in its own option name space) plus some SSL specific options:

- **proactive.pnps.port** : same as **proactive.pnp.port**
- **proactive.pnps.default_heartbeat** : same as **proactive.pnp.default_heartbeat**
- **proactive.pnps.idle_timeout** : same as **proactive.pnp.idle_timeout**
- **proactive.pnps.authenticate** : By default, PNPS only ciphers the communication but does not authenticate nor the client nor the server. Setting this option to true enable client and server authentication. If set to true the option **proactive.pnps.keystore** must also be set.
- **proactive.pnps.keystore** : Specify the keystore (containing the SSL private key) to use. If not set a private key is dynamically generated for this execution. The keystore must be created with the **ssl-keystore** script.

When using the authentication and ciphering mode (or the speed up the initialization in ciphering only mode) the option **proactive.pnps.keystore** must be set and a keystore embedding the private SSL key must be generated by using the **ssl-keystore** script.

To create the keystore, use the following command:

```
$ $PA_HOME/bin/ssl-keystore.sh --create --keystore mykeystore.p12
```

This keystore must be accessible to the runtimes and kept secret.

12.1.2.4. Troubleshooting

1. Runtimes hang when initializing

Using SSL consume require some available entropy to initialize the SSL engine. When the certificates are created at runtime (ciphering only mode without a keystore specified) SSL consumes even more entropy bytes. Most likely the runtime is waiting for more random bytes being available in the system entropy pool (/dev/random on UNIX). The most secure way to address this issue is to... way. If you don't want to wait until new bytes are available you can configure Java to use /dev/urandom instead of /dev/random . /dev/urandom is a little bit less secure than /dev/random, but it is non blocking.

12.1.3. ProActive Message Routing (PAMR, pamr://)

12.1.3.1. Description

The goal of PAMR is to allow the deployment of ProActive Programming when the configuration of the network is unfriendly. PAMR has the weakest expectations on how the network is configured. Unlike all the other communication protocols, PAMR does not expect

bidirectional TCP connections. It has been designed to work when only outgoing TCP connections are available. Such environments can be encountered due to:

- Network address translation devices
- Firewalls allowing only outgoing connection (this is the default setup of many personal firewall)
- Virtual Machines with a virtualized network stack

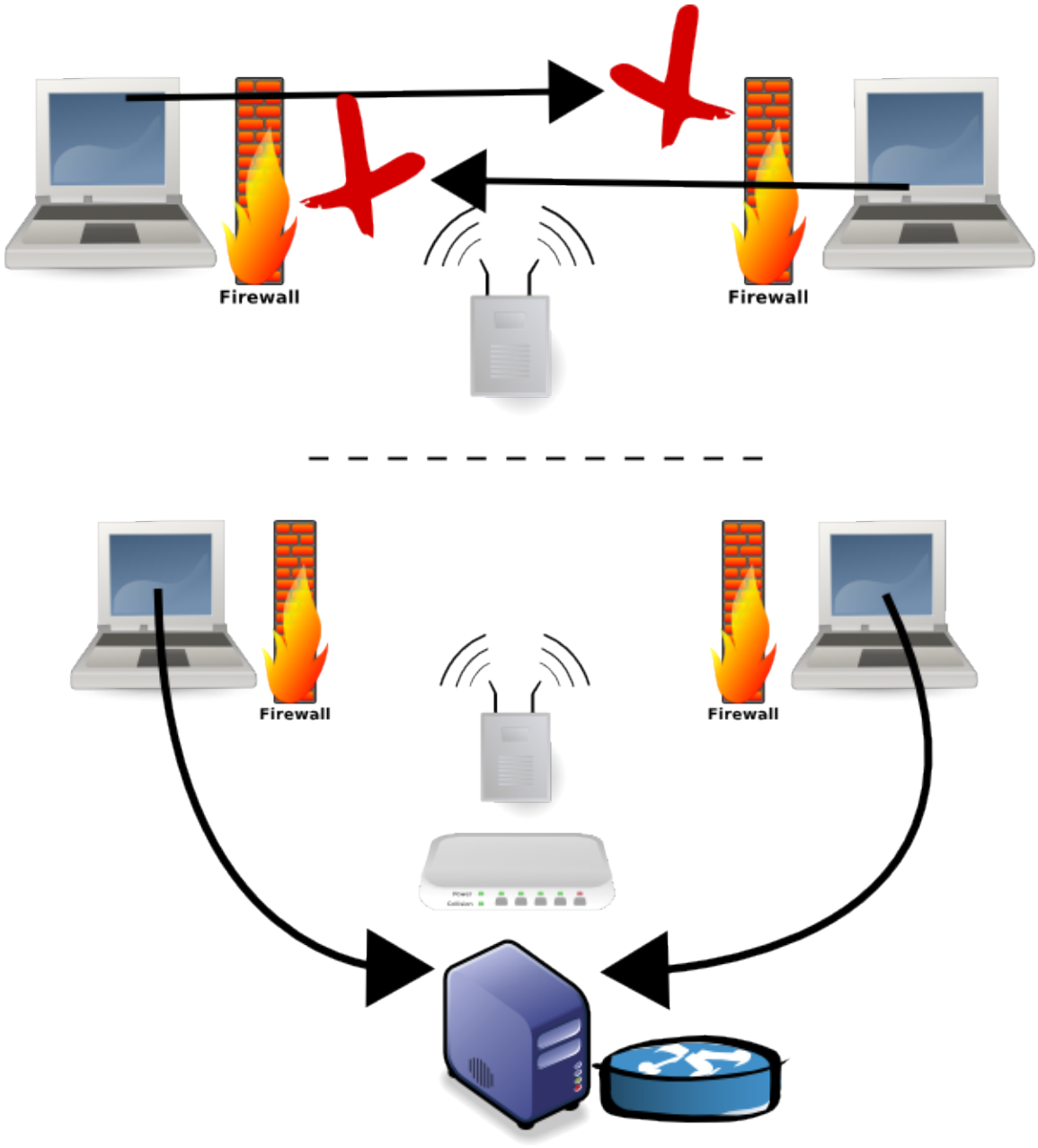
When PAMR is activated, each ProActive Programming runtime connect to a PAMR router. This connection is kept open, and used as a tunnel to receive incoming messages. If the tunnel goes down, it is automatically reopened by the ProActive runtime.

The biggest drawback of PAMR is that a centralized PAMR router is in charge of routing message between all the PAMR clients. To soften this limitation PAMR can be used with other communication protocol. This way, PAMR is used only when needed. See Enabling several communication protocols .

12.1.3.2. Use cases

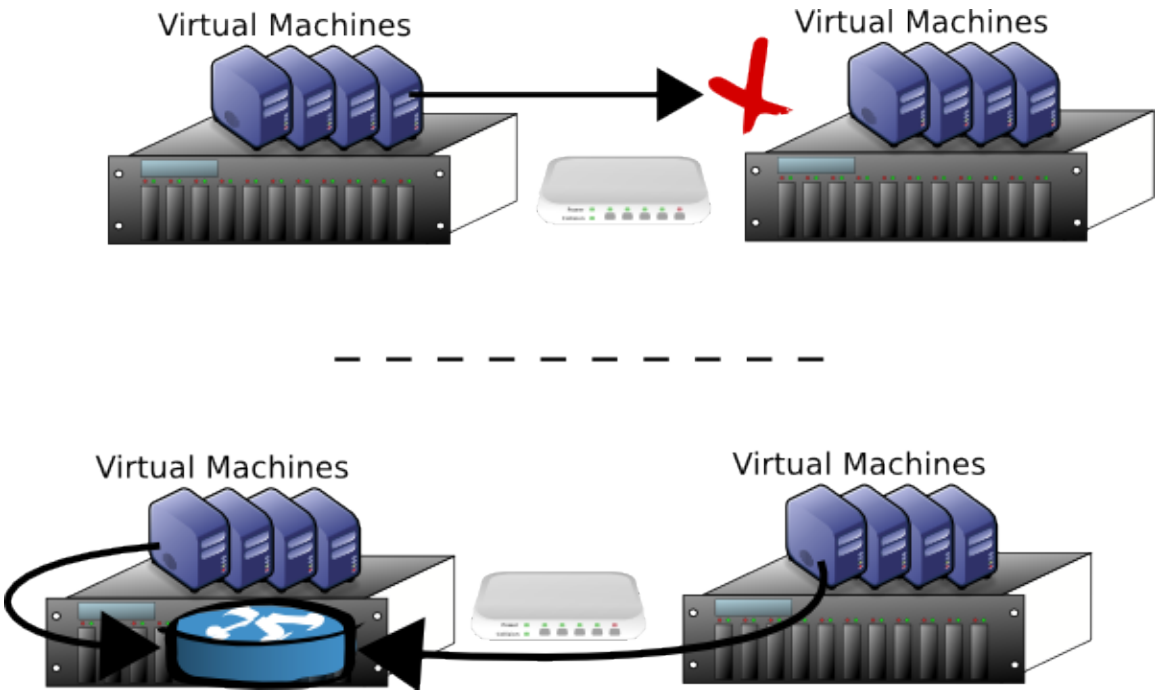
Few use cases of PARM follows.

The first use case is two laptops connected by WIFI, each of them using a personal firewall. As shown on the picture hereafter, there is no way to establish a connection between these two laptops. To solve this issue, a message router can be deployed on an external machine. Most of the desktop machines and laptop have a personal firewall enabled. If it is possible, it is better to open on TCP port on each machine and use the HTTP transport. If not, then message routing can be used.



Laptops with personal firewalls denying incoming connection.

The second use case is similar to the first one. When a virtual machine is deployed with a virtual network stack, it is not possible to contact the virtual machines from the LAN. VMs are behind a software NAT performed by the hypervisor.



Servers with few virtual machines deployed with a virtualized network stack.

The solution is to place a message router on a physical machine on the LAN. All the virtual machines will connect to this router. For production usage, it is better to configure your VM to get real IP addresses on the LAN. But with message routing you can develop and test your application on virtual machines without any configuration.

12.1.3.3. Configuration

The configuration of PAMR is a two step process. The first step is to configure and start the message router. The second step is to configure every ProActive runtime to use PAMR with the started message router.

12.1.3.3.1. Router configuration

The message router has to be started on a machine which is TCP reachable by all the ProActive runtimes involved in the computation. The localization of the message router is critical: the machine has to be TCP reachable, have a good network connection (bandwidth and latency) and be stable. If the message router crashes, the whole application must be restarted.

To start a message router, use the **bin/startRouter.sh** or **bin/startRouter.bat** script:

```
[cmathieu@britany:programming-git]$ bin/startRouter.sh
34980@britany.activeeon.com - [INFO forwarding.router] Message router listening on
ServerSocket[addr=/192.168.1.22,localport=33647]
```

The IP address and TCP port on which the message router is bound is printed on the standard output.

By default, the message router binds to the **wildcard** address on the **33647** TCP port. This behavior can be configuration by using the following parameters:

- **--port** : The TCP port to bind to.
- **--ip** : The IP address to bind to.
- **--ipv4** : The message router will only use IPv4 sockets
- **--ipv6** : Prefer IPv6 addresses over IPv4 addresses (and IPv4 addresses mapped over IPv6),

- **--nbWorkers** : The number of worker threads to use. The only serial operation in the message router is reading bits from the tunnels. All other operations can be performed in parallel with the help of a thread pool. This parameter defines how many workers are in the thread pool.
- **--configFile** : The configuration file to use to declare reserved clients. This file is a Java property file. Keys must be integers between 0 and 4095, value must be Unicode strings shorter than 64 characters. If a invalid value is found, the router will not start.
- **--timeout** : PAMR uses an heartbeat mecanism to detect network failures. As soon as an agent detects that the tunnel is broken, it will try to reconnect to the router. If the router detects a broken tunnel it sends a notification to every clients to unlock blocked thread waiting for a response from the disconnected client. This option sets the heartbeat timeout in milliseconds. It means that if the router or a client does not receive an heartbeat before the timeout, then the tunnel is closed and the client is disconnected. An heartbeat is sent every at an timeout/3 frequency.

  This value should not be set to a too low value. Value lower than 600ms on a LAN are unreliable (lot of false positive due to scheduling/network latency).

12.1.3.3.2. Well known urls

By default the router assigns an unique and random Agent ID to each agent. This random ID in the URL can make PAMR quite hard to use to deploy daemons and services. Indeed each time the services is restarted, a new URL is created.

To address this issue, PAMR provides reserved agent ID. This feature allows a runtime to ask for a specific agent ID. If an agent is already connected then it is disconnected and the slot is given to the new agent. Reserved agent ID must be declared in the router configuration file and a magic cookie must be specified for each reserved agent ID. This cookie is used to secure the slot, a client must provide the right magic cookie to be allowed to use the slot.

All reserved agent ID must be declared in a configuration file, and the router must be started with the **--configFile** option. The configuration file is a standard Java property file. Keys must be an integer between 0 and 4095 (the agent ID) and values are Unicode strings up to 64 characters.

A magic cookie for the key **configuration** must be set. It allows the administrator to reload the router configuration at runtime. See the **--reload** option.

```
$ cat services.txt
# Admin cookie
configuration=mySecretPassword
# Application 1
0=magicCookie1
# Application 2
2000=toor
2001=admin

$ ./bin/startRouter.sh --configFile services.txt
```

The **proactive.communication.pamr.agent.id** and **proactive.communication.pamr.agent.magic_cookie** ProActive properties must be used on the client runtime to configure the PAMR agent.

12.1.3.4. Troubleshooting

If the router cannot be contacted, an error message is printed with an exception like this one:

```
6743@britany.activeeon.com - [INFO oactive.remoteobject] Loading <pamr, class
 org.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory>
```

```
6743@britany.activeeon.com - [FATAL warding.remoteobject] Failed to
 initializeorg.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory
org.objectweb.proactive.core.ProActiveRuntimeException: Failed to create the local agent
 at
 org.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory.logAndThrowException(Mess
 at
 org.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory.<init>(MessageRoutingRemo
 at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
 at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
 at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
 at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
 at java.lang.Class.newInstance0(Class.java:355)
 at java.lang.Class.newInstance(Class.java:308)
 at
 org.objectweb.proactive.core.remoteobject.AbstractRemoteObjectFactory.getRemoteObjectFactory(AbstractRemoteObjectFactory
 at
 org.objectweb.proactive.core.remoteobject.AbstractRemoteObjectFactory.getDefaultRemoteObjectFactory(AbstractRemoteObject
 at
 org.objectweb.proactive.core.remoteobject.RemoteObjectExposer.createRemoteObject(RemoteObjectExposer.java:153)
 at org.objectweb.proactive.core.runtime.ProActiveRuntimeImpl.<init>(ProActiveRuntimeImpl.java:267)
 at org.objectweb.proactive.core.runtime.ProActiveRuntimeImpl.<clinit>(ProActiveRuntimeImpl.java:158)
 at
 org.objectweb.proactive.extensions.gcmdeployment.GCMApplication.NodeMapper.subscribeJMXRuntimeEvent(NodeMapper.java
 at org.objectweb.proactive.extensions.gcmdeployment.GCMApplication.NodeMapper.<init>(NodeMapper.java:97)
 at
 org.objectweb.proactive.extensions.gcmdeployment.GCMApplication.GCMApplicationImpl.<init>(GCMApplicationImpl.java:176)
 at
 org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment.loadApplicationDescriptor(PAGCMDeployment.java:104
 at
 org.objectweb.proactive.extensions.gcmdeployment.PAGCMDeployment.loadApplicationDescriptor(PAGCMDeployment.java:86)
 at org.objectweb.proactive.examples.hello.Hello.main(Hello.java:87)
Caused by: org.objectweb.proactive.core.ProActiveException: Failed to create the tunnel to
 britany.activeeon.com/192.168.1.22:33647
 at org.objectweb.proactive.extra.messagerouting.client.AgentImpl.<init>(AgentImpl.java:128)
 at org.objectweb.proactive.extra.messagerouting.client.AgentImpl.<init>(AgentImpl.java:88)
 at
 org.objectweb.proactive.extra.messagerouting.remoteobject.MessageRoutingRemoteObjectFactory.<init>(MessageRoutingRemo
 ... 17 more
```

The nested exception shows that the ProActive runtime failed to contact the message router.

12.1.4. Java RMI

12.1.4.1. Description

RMI is the default communication protocol. It combines good performances with remote class loading. Drawbacks of RMI are that by default all the runtimes on a same machine share the same RMI registry, there is no network failure detection mechanism other than the TCP timeout, poor scalability due to a huge number of threads being created, and not firewall friendly.

RMI is the protocol to use if you deploy your application on an non firewalled LAN, don't need high scalability, and want best point to point performance (both bandwidth and throughput).

12.1.4.2. Configuration

- **proactive.rmi.port** : The TCP port to use for the RMI registry. Cannot be 0 for dynamic port selection. RMI uses other random TCP ports for the communication, this port is only used by the RMI registry.

- **proactive.rmi.connect_timeout** : The amount of time in millisecond to way wait while establishing a TCP connection before reporting it as failed. This option allows to not wait the TCP timeout when a machine is firewalled and drop the packets (default Windows firewall behavior).

12.1.5. HTTP

12.1.5.1. Description

This communication protocol allows to use an embedded Jetty server to communicate by using HTTP message. HTTP is more firewall friendly than RMI since it only uses one TCP port.

When possible, it is better to use PNP than HTTP since PNP provides more features like fast network failure discovery or SSL encryption. In addition PNP is usually faster than HTTP. Nevertheless HTTP is useful to pass through layer 7 filtering since the HTTP protocol will be most likely be allowed.

12.1.5.2. Description

- **proactive.http.port** : The TCP port to bind to. If not set HTTP uses a random free port. If the specified TCP port is already used HTTP will not start and an error message is displayed.
- **proactive.http.jetty.connector** : The Connector to be used by Jetty (full class name). By default a SelectChannelConnector is used. It is well suited to handle a lot of mainly idle clients workload (like coarse grained master worker). It can be replaced by a Socketconnecter to achieve better performances for an application with a few very busy client. SocketConnect, BlockingChannelConnector and SelectChannelConnector can be used. See Jetty documentation[1] for more information.
- **proactive.http.jetty.xml** : File path to Jetty configuration to use. See Jetty documentation[2] for more information
- **proactive.http.connect_timeout** : The HTTP socket timeout in milliseconds. A timeout of zero is interpreted as an infinite timeout.

12.2. TCP/IP configuration

ProActive programming is only able to bind to one and only one InetAddress. Usually, this limitation is not seen by the user and no special configuration is required. ProActive tries to use the most suitable network address available. But sometimes, ProActive fails to elect the right IP address or the user wants to use a given IP address. In such case, you can specify the IP address to use by using theses properties: **proactive.hostname** , **proactive.net.interface** , **proactive.net.netmask** , **proactive.net.nolocal** , **proactive.net.noprivate** .

IPv6 can be enabled by setting the **proactive.net.disableIPv6** property to **false** . By default, ProActive does not use IPv6 addresses.

If none of the **proactive.hostname** , **proactive.net.interface** , **proactive.net.netmask** , **proactive.net.nolocal** , **proactive.net.noprivate** properties is defined, then the following algorithm is used to elect an IP address:

- If a public IP address is available, then use it. If several ones are available, one is randomly chosen.
- If a private IP address is available, then use it. If several ones are available, one is randomly chosen.
- If a loopback IP address is available, then use it. If several ones are available, one is randomly chosen.
- If no IP address is available at all, then the runtime exits with an error message.

If **proactive.hostname** is set, then the value returned by **InetAddress.getByName(proactive.hostname)** is elected. If no IP address is found, then the runtime exits with an error message.

If **proactive.hostname** is not set, and at least one of the **proactive.net.interface** , **proactive.net.netmask** , **proactive.net.nolocal** , **proactive.net.noprivate** is set, then one of the addresses matching all the requirements is elected. Requirements are:

- If **proactive.net.interface** is set, then the IP address must be bound to the given network interface.
- If **proactive.net.netmask** is set, then the IP address must match the given netmask.
- If **proactive.net.nolocal** is set, then the IP address must not be a loopback address.

[1] http://docs.codehaus.org/display/JETTY/Architecture
[2] http://docs.codehaus.org/display/JETTY/Syntax+Reference

- If **proactive.net.noprivate** is set, then the IP address must not be a private address.

If no address matches theses criteria, the runtime exits with an error message.

The easiest way to check if ProActive elect the address you want is to run the following command:

```
java -jar dist/lib/ProActive.jar
```

It shows the elected IP address and all the network interfaces and IP addresses available. It also displays all the configuration properties and their value.

```
--------------------
ProActive 2008-07-10 11:59:18
--------------------


Local IP Address: 192.168.1.22
Config dir: /user/cmathieu/home/.proactive

Network setup:
eth0 MAC n/a fe80:0:0:0:222:19ff:fe1d:ad34%2 192.168.1.22
lo MAC n/a 0:0:0:0:0:0:0:1%1 127.0.0.1

Available properties:
String java.security.policy [null]
Boolean java.net.preferIPv6Addresses [null]
```

Here are some examples you can adapt to your situation:

```
$ # Use the default algorithm, since no public IP address is available, 192.168.1.22 is elected.
$ java -jar dist/lib/ProActive.jar

$ # Use the IP address returned by name resolution system for "britany.activeeon.com"
$ java -Dproactive.hostname=britany.activeeon.com -jar dist/lib/ProActive.jar

$ # Use only a public IP address bound to eth1
$ java -Dproactive.net.interface=eth1 -Dproactive.net.nolocal=true -Dproactive.net.noprivate=true -jar dist/lib/
ProActive.jar

$ # Use an IP address matching 192.168.2.0/24
$ java -Dproactive.net.netmask="192.168.2.0/24" -Dproactive.net.noprivate=true -jar dist/lib/ProActive.jar
```

The log4j logger **proactive.configuration.network** can be used to diagnosis why an IP address is elected or rejected.

12.3. Enabling several communication protocols

Chapter 13. Using SSH tunneling for RMI or HTTP communications

13.1. Overview

ProActive allows users to **tunnel** all of their RMI or HTTP communications over **SSH**: it is possible to specify into the 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 connections 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 objects makes requests to which objects**. For example: A -> B, B -> C, A ->C

13.2. Network Configuration

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

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

If you use a TCP or a UDP-based communication protocol (RMI is based on TCP), these questions can be translated into 'what **ports** on B **allows** A 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:

```
 Firewall                  Firewall
    |   *                 *   |
    | ---->    Internet     <---- |
A   | <----                 ----> |   B
    |   22                  22   |
```

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.

13.3. 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.

13.4. 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 objects connects to what objects to identify the influence of the network configuration on these communication patterns.

13.5. 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 unique property named **proactive.communication.protocol** which is set to one of:

- rmi
- http
- rmissh (RMI over SSH)
- rmissl (RMI over SSL)
- ibis
- pamr (plain TCP connection or tunneled over SSH)

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 **GCM Application Descriptor** and to pass the property as a command-line option to the JVM:

```
<application>
  <proactive base="root" relpath="${proactive.home}">
    <configuration>
      <jvmarg value="-Dproactive.communication.protocol=rmissh" />
  </proactive>
</application>
```

  To know more about GCM deployment, please refer to Chapter 14, *ProActive Grid Component Model Deployment*.
- the other one is to set, in the **ProActive Configuration file** (introduced in Chapter 11, *ProActive Basic Configuration*) 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.

13.6. 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 to 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 chosen by the ssh tunnel. The ssh daemon running on the server host receives the data for this tunnel, removes its encapsulation 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 **ganymed** 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.communication.rmissh.port**: Port number on which all the ssh daemons to which this JVM has to connect to are expected to listen. The default value is **22**.
- **proactive.communication.rmissh.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 will not be parsed properly.

  If this property is not set, the default is the **user.name** Java property.
- **proactive.communication.rmissh.known_hosts**: 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. The default is **System.getProperty('user.home') + '/.ssh/known_hosts'**
- **proactive.communication.rmissh.key_directory**: 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 has to be suffixed by '.pub'. Private keys are ignored if their associated public key is not present. The default is **System.getProperty('user.home') + '/.ssh/'**.
- **proactive.communication.rmissh.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 The default is **no**, meaning these direct-connection will not be attempted. 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 front-end 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 front-end.
- **proactive.communication.rmissh.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. The default value is **2000 ms**.
- **proactive.communication.rmissh.gc_idletime**: This property identifies the maximum idle time before a SSH tunnel or a connection is garbage collected.
- **proactive.communication.rmissh.gc_period**: This property specifies how long the tunnel garbage collector will wait before destroying an unused tunnel. If a tunnel is older than this value, it is automatically destroyed. The default value is **10000 ms**.

Note that the use of SSH tunneling over RMI still allows dynamic class loading through HTTP. For the dynamic class loading, 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.communication.protocol** property to **rmissh** and to use the related properties if needed (in most cases, default behavior is sufficient), ProActive takes care of everything else.

Part IV. Deployment And Virtualization

Table of Contents

Chapter 14. ProActive Grid Component Model Deployment

14.1. Introduction

This chapter is meant to explain the GCM deployment concept and to provide you with all the necessary knowledge to be able to deploy your applications. The GCM Deployment is split in two parts: one for grid administrators and the other for grid application developers. On the grid administration side, the administrator will write a Deployment Descriptor that will describe what resources the grid provides, and how these resources are acquired. On the application side, the developer will write an Application Descriptor that will describe how the application is launched, and what resources it needs. The link between the two sides is made through references from the Application Descriptor to one or several Deployment Descriptors.

Before the GCM Deployment, ProActive used another deployment structure which is still supported. To read about the deprecated deployment structure, please refer to Chapter 15, *XML Deployment Descriptors*.

14.2. Deployment Concepts

A first principle is to fully eliminate from the source code the following elements:

- machine names
- creation protocols
- registry lookup protocols

The goal is to deploy any application anywhere without changing the source code. For instance, we must be able to use various protocols, rsh, ssh, Globus, LSF, etc. for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as RMIregistry, Globus etc. Therefore, we see that the creation, registration and discovery of resources have to be done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. The description should indicate the various parallel or distributed entities in the program. For instance, an application that is designed to use three interactive visualization nodes, a node to capture input from a physics experiment, and a simulation engine designed to run on a cluster of machines should somewhere clearly advertise this information.

However, it important to keep in mind that the abstract description of an application and the way to deploy it are not independent pieces of information. If for example, we have a simulation engine, it might register in a specific registry protocol, and if so, the other entities of the computation might have to use that lookup protocol to bind to the engine. Moreover, one part of the program can just lookup for the engine (assuming it is started independently), or explicitly create the engine itself. To summarize, in order to abstract away the underlying execution platform, and to allow a source-independent deployment, a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component.
- an external mapping of those entities to real machines, using actual creation, registry, and lookup protocols.

To reach that goal, the programming model relies on the specific notion of Virtual Nodes (VNs):

- a VN is identified by a name (a simple string)
- a VN is used in a program source
- a VN is defined and configured in a deployment descriptor (XML format)
- a VN, after activation, is mapped to one or to a set of actual ProActive Nodes

Of course, distributed entities (Active Objects), are created on Nodes, not on Virtual Nodes. There is a strong need for both Nodes and Virtual Nodes. Virtual Nodes are a much richer abstraction, as they provide mechanisms such as set or cyclic mapping. Another key

aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical if we want to get at once machines from a cluster of PCs managed through Globus or LSF. It is even more critical in a Grid application, when trying to achieve the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting Active Objects. There is obviously a correspondence between Virtual Nodes and Nodes: the function created by the deployment, that is to say, the mapping. This mapping is specified in the Application Descriptor. The grid facilities are described in two deployment descriptor separated by the different concerns of the application developer and grid infrastructure administrator.

In the grid deployment descriptor, we describe:

- the resources provided by the infrastructure
- how to acquire the resources provided by the infrastructure

As for the application deployment descriptor, we describe:

- how to launch the application
- the resources needed by the application
- the resource providers

Here is a picture that represents the deployment architecture and especially the relations between application, virtual node, application descriptor and deployment descriptor:

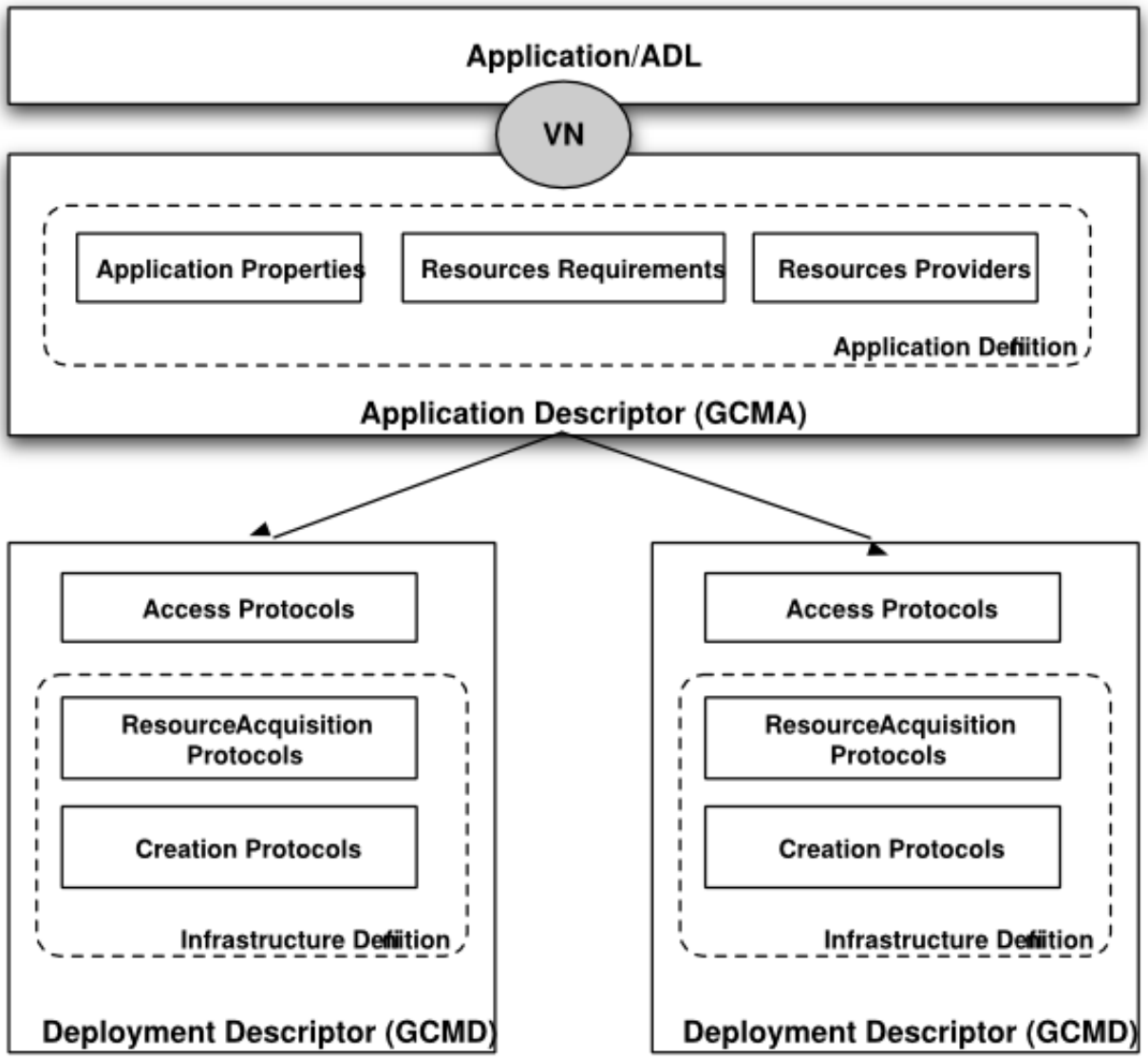


Figure 14.1. Deployment architecture

The following figure illustrates a simplified view of the deployment process. The ProActive application loads the deployment descriptor and deploys on the remote machine according to the settings in the descriptor. Although the process behind the deployment is fairly complicated, it is made seamless by ProActive. In the application, we only need to specify the application descriptor and tell ProActive to start virtual nodes, nodes and active objects. The communication details are handled by ProActive according to the descriptor.

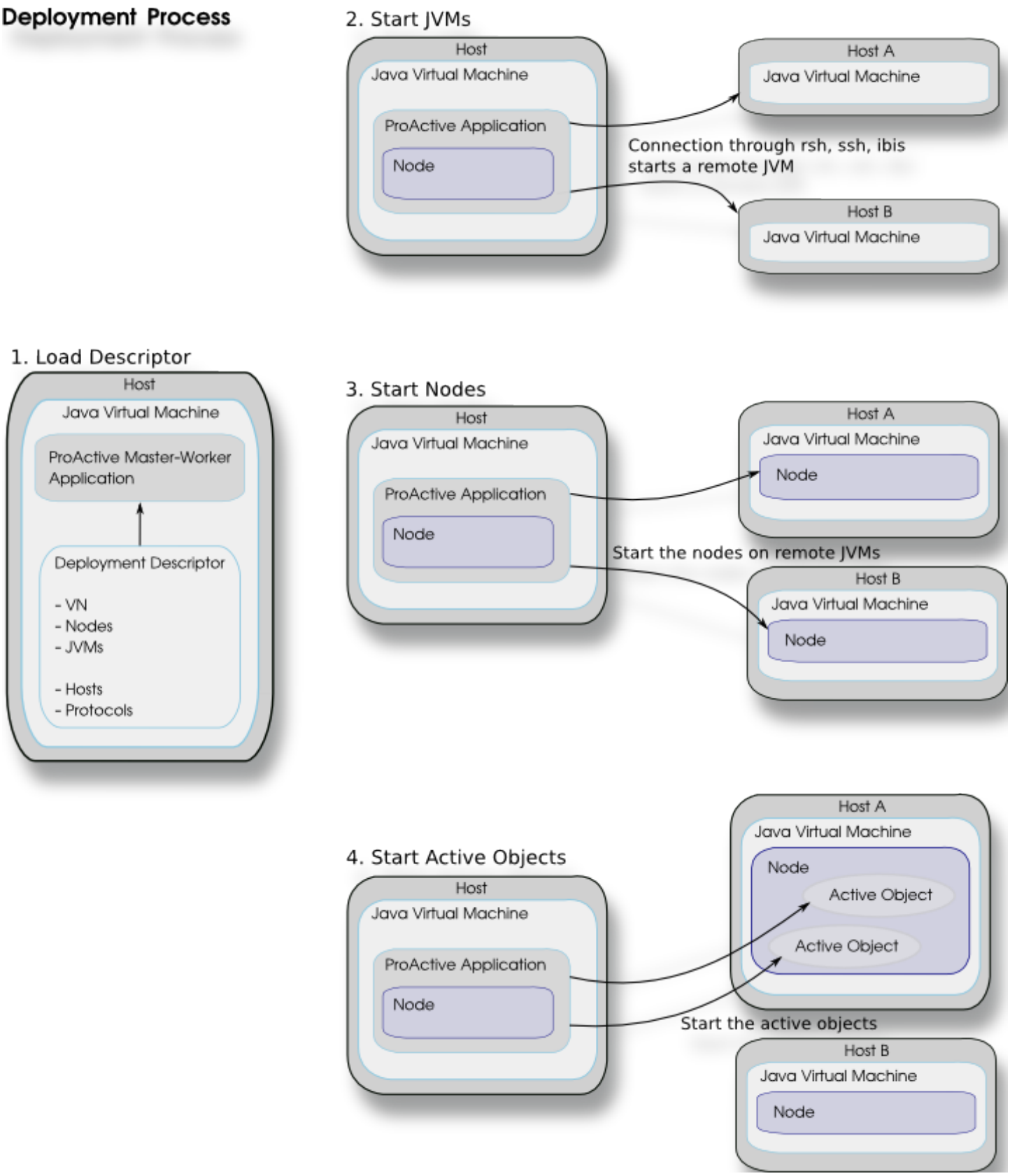


Figure 14.2. Deployment process

The whole deployment process and environment configuration is defined by means of XML descriptors which depict the application requirements and deployment process. The deployment of ProActive/GCM applications depends on two types of descriptors:

- The GCM Application Descriptors (GCMA): the GCMA descriptors define applications-related properties, such as localization of libraries, file transfer, application parameters and non-functional services (logging, security and checkpoint). The resource requirement is also defined, but taking the VNs and nodes into account. Besides, the GCMA defines one or multiple resource providers.
- The GCM Deployment Descriptors (GCMD): the GCMD descriptors define the operation of the resource providers. This includes the access protocols to reach the resources (e.g. SSH, RSH, GSISSH, etc.), acquisition protocols and tools which are sometimes required to have access to resources (e.g. PBS, LSF, Sun Grid Engine, OAR, etc.), creation protocols which have a relation on how to launch processes (e.g. SSH, OAR, gLite, Globus) and communication protocols (e.g. RMI, RMISSH, HTTP, SOAP, etc).

Overviews of these two descriptors are exposed respectively in Section 14.3, “GCM Deployment Descriptor Overview” and in Section 14.4, “GCM Application Descriptor Overview”. There are also completely described in Section 14.6, “GCM Deployment Descriptor”.

The need of this two kinds of descriptor enforces a clear separation between application definition and deployment process. The advantages of this model are clear: if, in one side, users want to add a new resource provider (e.g. a private cluster, production grid or cloud), the application code does not change and a single line is enough to add the resource provider to the application descriptor (GCMA). On the other side, the definition of the deployment process happens just once for each resource and can be reused for different applications.

14.3. GCM Deployment Descriptor Overview

The deployment descriptor is an XML file containing information on the properties listed above. We will use a simple XML file to deploy our applications on a remote machine. The deployment XML file is composed of several parts, each with different options. In this section, we will describe a simple version. To find out more about deployment and deployment descriptors, please refer to Section 14.6, “GCM Deployment Descriptor” which strives to give an exhaustive list of possible tags.

The document uses the XML Schema [1] present at the Oasis website.

To avoid mistakes when building XML descriptors, ProActive provides two XML Schemas, one for each descriptor type.

To validate your deployment descriptor file, the following line has to be put at the top of the XML document.

```
<GCMDeployment xmlns="urn:gcm:deployment:1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:deployment:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ExtensionSchemas.xsd">
```

Both XML files have a section for defining variables needed later in the document. For instance, the following block defines the user home:

```
<environment>
  <javaPropertyVariable name="user.home" />
</environment>
```

Let's take a closer look at the deployment descriptor. We first need to specify how the grid resources are organized together. This is done in the resources section, in which we describe a tree-like structure corresponding to the grid setup. There are three types of elements:

- host: a single machine
- bridge: a gateway to a set of machines which cannot be reached individually
- group: a group of machine sharing an identical configuration (which is described by a host)

```
<resources>
```

[1] http://proactive.inria.fr/schemas/gcm/1.0/ExtensionSchemas.xsd

```
  <group refid="rshLan">
    <host refid="ComputeNode" />
  </group>
</resources>
```

Next is the infrastructure part that defines the elements (hosts, bridges, groups) which are referenced in the resources part. In the following example, hosts tag defines only the home directory and the operating system. Yet, you can also define the java home or the machine workload capacity. In a general manner, bridges and groups tags define what kind of configuration they are. Here, we have a group of machines accessible through rsh. This group is then described in detail into the rshGroup tag where the hostList attribute references all the host names.

```
<infrastructure>
  <hosts>
    <host id="ComputeNode" os="unix" hostCapacity="1">
      <homeDirectory base="root" relpath="${user.home}" />
    </host>
  </hosts>

  <groups>
    <rshGroup id="rshLan" hostList="kisscool paquito jily sgouirk" />
  </groups>
</infrastructure>
```

Thus, here is full and simple deployment descriptor:

```
<GCMDeployment xmlns="urn:gcm:deployment:1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:deployment:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ExtensionSchemas.xsd">

  <environment>
    <javaPropertyVariable name="user.home" />
  </environment>

  <resources>
    <group refid="rshLan">
      <host refid="ComputeNode" />
    </group>
  </resources>

  <infrastructure>
    <hosts>
      <host id="ComputeNode" os="unix" hostCapacity="1">
        <homeDirectory base="root" relpath="${user.home}" />
      </host>
    </hosts>

    <groups>
      <rshGroup id="rshLan" hostList="kisscool paquito jily sgouirk" />
    </groups>
  </infrastructure>
</GCMDeployment>
```

14.4. GCM Application Descriptor Overview

Now, let's study application descriptors where we define the application and its requirements.

In the same manner as for the deployment descriptor, the following line enables to validate our descriptor according to its schema:

```
<GCMApplication xmlns="urn:gcm:application:1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">
```

Like a deployment descriptor, an application descriptor starts with an environment section, which follows the same syntax. The following block defines for example the proactive home as well as two descriptor variables which are respectively the host capacity (number of JVMs per host) and the vmCapacity (number of nodes per JVM).

```
<environment>
  <javaPropertyVariable name="proactive.home" />
  <javaPropertyVariable name="user.home" />
  <descriptorVariable name="hostCapacity" value="1"/>
  <descriptorVariable name="vmCapacity" value="1"/>
</environment>
```

Then, there is the application section in which the application itself is described: its type (ProActive or stand-alone executable), its configuration (dependencies, invocation options) and its resource requirements (the virtual nodes). In this example, we have defined a single virtual node named 'Agent'. Virtual nodes refer to node providers, which are defined in the next section, as sources of physical nodes.

```
<application>
  <proactive base="root" relpath="${proactive.home}">
    <configuration>
      <applicationClasspath>
        <pathElement base="proactive" relpath="dist/lib/ProActive_examples.jar"/>
        <pathElement base="proactive" relpath="dist/lib/ibis-1.4.jar"/>
        <pathElement base="proactive" relpath="dist/lib/ibis-connect-1.0.jar"/>
        <pathElement base="proactive" relpath="dist/lib/ibis-util-1.0.jar"/>
      </applicationClasspath>
    </configuration>
    <virtualNode id="Agent" capacity="4">
      <nodeProvider refid="RSHNodeProvider" />
    </virtualNode>
  </proactive>
</application>
```

Finally, we link the application descriptor to one or several deployment descriptors through its resources part. Into this part, you can define the node providers that are referenced by the virtual nodes in the previous section.

```
<resources>
  <nodeProvider id="RSHNodeProvider">
    <file path="RSHLan.xml"/>
  </nodeProvider>
</resources>
```

The whole application descriptor therefore looks as follows:

```
<GCMApplication xmlns="urn:gcm:application:1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">

  <environment>
    <javaPropertyVariable name="proactive.home" />
    <javaPropertyVariable name="user.home" />
```

```
    <descriptorVariable name="hostCapacity" value="1"/>
    <descriptorVariable name="vmCapacity" value="1"/>
  </environment>

  <application>
    <proactive base="root" relpath="${proactive.home}">
      <configuration>
        <applicationClasspath>
          <pathElement base="proactive" relpath="dist/lib/ProActive_examples.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-1.4.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-connect-1.0.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-util-1.0.jar"/>
        </applicationClasspath>
      </configuration>
      <virtualNode id="Agent" capacity="4">
        <nodeProvider refid="RSHNodeProvider" />
      </virtualNode>
    </proactive>
  </application>

  <resources>
    <nodeProvider id="RSHNodeProvider">
      <file path="RSHLan.xml"/>
    </nodeProvider>
  </resources>

</GCMApplication>
```

14.5. ProActive Deployment API

There are several ways the grid resources can be used by a deployed application. The application may require a fixed set of resources, or it may be flexible enough to work on any amount of resources, or finally may require a minimum amount of resources and yet be able to scale as more resources become available.

In all cases, the application must start by creating a GCMApplication object through PAGCMDeployment.loadApplicationDescriptor(), and call GCMApplication.startDeployment(). The application must quit through GCMApplication.kill().

```
// Retrieves the file corresponding to your application descriptor
File applicationDescriptor = new File(descriptorPath);

GCMApplication gcmad;
try {

  // Loads the application descriptor file
  gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);

  // Starts the deployment
  gcmad.startDeployment();

  // ...

  // Terminates the deployment
  gcmad.kill();
```

```
} catch (ProActiveException e) {
    e.printStackTrace();
}
```

14.5.1. Resources fixed by the application (SPMD)

In this case, the application knows the amount of resources it requires. The acquisition of these resources by the application is done as follows:

- Get the required virtual nodes through GCMApplication.getVirtualNode(String vnName), or GCMApplication.getVirtualNodes()
- For each virtual node, use GCMVirtualNode.getCurrentNodes() as many times as needed, until the virtual node has the expected numbers of physical nodes to run on. getCurrentNodes() will return the list of nodes that have been acquired since the last time it was called. Calls to it should be separated by calls to Thread.sleep().

```
// Loads the application descriptor file
gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);

// Starts the deployment
gcmad.startDeployment();

GCMVirtualNode vn = gcmad.getVirtualNode("Agent");

List<Node> nodeList = vn.getCurrentNodes();
while (nodeList.size() < 3) {

    Thread.sleep(2000);
    nodeList = vn.getCurrentNodes();
}
```

14.5.2. Resources fixed by the application deployer

In this case, the application has no specific requirement on the resources it uses: the more the better. This is the simplest case: the application only has to call GCMApplication.waitReady(). This will block until all Virtual Nodes have their configured number of physical Nodes. In our case, we have only one virtual node (named Agent) whose capacity is 4. Thus, our application will block until Agent gets its 4 nodes.

```
// Loads the application descriptor file
gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);

// Starts the deployment
gcmad.startDeployment();

GCMVirtualNode vn = gcmad.getVirtualNode("Agent");
vn.waitReady();
```

Warning

This may block forever if a Virtual Node does not have a limited number of nodes after which it is in 'ready' state (the Virtual Node is said to be 'greedy', GCMVirtualNode.isGreedy() will return true).

14.5.3. On demand Scalability

In this case, the application is able to expand on new resources as they become available. This is an extension of the two other cases, in that it can work whether the application has fixed minimum requirements or not. Once the initial deployment phase is finished,

the application should call GCMApplication.getVirtualNodes() to obtain the list of configured virtual nodes, and then subscribe to the node attachment notifications for each of them (GCMVirtualNode.subscribeNodeAttachment()). In the notification handler, the application should deal with the newly acquired node appropriately.

```
// Loads the application descriptor file
gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);

// Starts the deployment
gcmad.startDeployment();

GCMVirtualNode vn = gcmad.getVirtualNode("Agent");
vn.subscribeNodeAttachment(this, "nodeAttached", false);

// Waiting for new nodes
Thread.sleep(5000);
```

14.6. GCM Deployment Descriptor

This section aims at giving explanations on the structure of a GCM Deployment descriptor. We will not describe all the possible elements and all the possible attributes since a Javadoc-like documentation [2] is generated directly from the XSD schema (depicting the XML structure) and is therefore up to date at any time.

As already said into the overview chapter, a GCM deployment descriptor is composed of 3 elements:

- **<environment>**
- **<resources>**
- **<infrastructure>**

The following sections will details these three elements.

14.6.1. The <environment> element

In the first part (<environment>), we can define environment variables such as the Java home, the ProActive home or a JVM argument... There are height different kinds of variable:

- **<descriptorDefaultVariable>**
- **<descriptorVariable>**
- **<includePropertyFile>**
- **<javaPropertyDescriptorDefault>**
- **<javaPropertyProgramDefault>**
- **<javaPropertyVariable>**
- **<programDefaultVariable>**
- **<programVariable>**

Here is a brief example showing how to defined a "descriptorVariable" variable:

```
<environment>
  <descriptorVariable name="myVariable" value="myValue"/>
</environment>
```

As a chapter is dedicated for variables, please refer to Chapter 18, *Variable Contracts for Descriptors* or directly to the Javadoc-like documentation [3] if you want more information.

[2] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcmd/index.html
[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcmd/index.html

Note

Such an element is also used by the GCM Application descriptor. It is therefore important to keep in mind that environment variables are inherited from the GCM Application descriptor.

14.6.2. The <resources> element

The second part (<resources>) is used to represent the grid architecture. It is a **tree-like structure into which we only specify ids** that will be used in the <infrastructure> part for describing each resource. If an id is used, it has to correspond to an existed id in the <infrastructure> element. If not, the XML file will not be valid.

For instance, we can make reference to a single host as follows:

```
<resources>
  <host refid="myHost" />
</resources>
```

Or we can describe a group of hosts. In this case, two ids are necessary: one for defining the group itself (group type, hosts list...) and one for characterizing the host configuration (all the hosts of a group have the same configuration).

```
<resources>
  <group refid="myGroup">
    <host refid="hostsOfMyGroup" />
  </group>
</resources>
```

Finally, we can define a bridge giving access to a host, a group of hosts or another bridge. The following example shows how to specify a bridge towards a group of hosts. In that case, three ids are needed: one for each element (bridge, group and host).

```
<resources>
  <bridge refid="myBridge">
    <group refid="groupAccessibleThroughMyBridge">
      <host refid="hostsOfMyGroupAccessibleThroughMyBridge"/>
    </group>
  </bridge>
</resources>
```

To get more information on this part, please refer to the Javadoc-like schema documentation [4].

14.6.3. The <infrastructure> element

In this part, all the elements referenced into the <resources> element are described. It is composed of three optional elements: **<hosts>** into which hosts configuration are specified, **<groups>** into which groups are defined (depending on the group type) and **<bridges>** into which bridge are depicted (also depending on the bridge type).

Warning

These elements are optional from the point of view of the <infrastructure> element. However, if you have made reference to a host in the <resources> element, you have to describe it into a <hosts> element. Otherwise, the GCMD descriptor would not be valid. This feature also holds for <groups> and <bridges> elements.

Host definition is quite simple. The important thing to remind is that an **"id"** as well as a **"os"** have to be provided concerning attributes. As for child elements, **<homeDirectory>** is the only mandatory element and it represents the path to the home directory. It is also

[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcmd/index.html

interesting to point out that the host capacity (number of JVMs per host) and the JVM capacity (number of nodes per JVMs) can be specify through respectively the "hostCapacity" and "vmCapacity" attributes.

Concerning groups, there are ten different group type:

- **<cssGroup>**
- **<gLiteGroup>**
- **<gridEngineGroup>**
- **<loadLevelerGroup>**
- **<lsfGroup>**
- **<oarGroup>**
- **<pbsGroup>**
- **<prunGroup>**
- **<rshGroup>**
- **<sshGroup>**
- **<mpiGroup>**

All these groups are explained in the schema documentation [5]. However, they all have two common attributes and one common child. Indeed, each group has an **"id"** attribute (used in the <resources> element) and a **"commandPath"** attribute (path of the command which is used to submit a job to the group protocol). The common child is an **<environment>** element which is of the same type as the one seen before and is used for defining variables specific to the group.

As for bridges, they are three different bridge type:

- **<rshBridge>**
- **<sshBridge>**
- **<oarshBridge>**

Only one attribute is common: the "id" attribute.

Here is a possible way to detail the hosts, groups and bridges previously defined in the <resources> element:

```
<infrastructure>
  <!-- Hosts Description -->
  <hosts>
    <host id="myHost" os="unix" hostCapacity="2">

      <!-- Mandatory element -->
      <homeDirectory base="root" relpath="/home" />

      <!-- Optional element -->
      <networkInterface name="eth0"/>
    </host>

    <host id="hostsOfMyGroup" os="unix">
      <!-- Mandatory element -->
      <homeDirectory base="root" relpath="/home" />
    </host>

    <host id="hostsOfMyGroupAccessibleThroughMyBridge" os="unix" hostCapacity="2" vmCapacity="3">
      <!-- Mandatory element -->
      <homeDirectory base="root" relpath="/home" />
    </host>
```

[5] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcmd/index.html

```
  </hosts>

  <groups>
    <rshGroup id="myGroup" hostList="host1 host2 host3" />

    <sshGroup id="groupAccessibleThroughMyBridge" hostList="host3 host4">
      <privateKey base="root" relpath="/home/.ssh/id_rsa"/>
    </sshGroup>
  </groups>

  <bridges>
    <sshBridge  id="myBridge" hostname="brideHost" username="ProActiveTeam" />
  </bridges>
</infrastructure>
```

To get more information on this part and specially on the different group and bridge type, please refer to the schema documentation [6].

14.7. GCM Application Descriptor

This section aims at giving explanations on the structure of a GCM Application descriptor. We will not describe all the possible elements and all the possible attributes since a javadoc-like schema documentation [7] is generated directly from the XSD schema (depicting the XML structure) and is therefore up to date at any time.

As already said into the overview chapter, a GCM application descriptor is composed of 3 elements:

- **<environment>**
- **<application>**
- **<resources>**

The following sections will details these three elements.

14.7.1. The <environment> element

This elements is exactly the same as those presented in Section 14.6.1, “The <environment> element”. As evoked before, environment variable define here will be inherited to the GCM Deployment descriptor.

14.7.2. The <application> element

This mandatory element is used for describing the application itself. It is composed of only one child element which can be either **<proactive>** or **<executable>**.

14.7.2.1. Executable

This type of application describes the deployment of a stand-alone executable on the grid. The chapter Chapter 9, *Wrapping Native MPI Application* shows a couple of examples of the usage of this application type

- **nodeProvider**: empty element with a single **"refId"** attribute representing the id of a node provider (defined in the <resources> part). There can be any number of such element.
- **command**: command which will be run on the portion of the grid defined by the specified node providers. It can have the following children (in this specified order):
  - **<path>**: the path of the executable
  - **<arg>**: the arg string which will be passed to the command. There can be any number of such elements.

  And it can have the following attribute:

[6] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcmd/index.html
[7] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../schemas/gcma/index.html

- **"name"**: name of the executable. If a <path> child element is present, the value of this attribute will be appended to the value of the <path> child element.

The <executable> element can also have the following attribute:

- **instances**: number of instances of the command which will be run. It can be “onePerHost”, “onePerVM” or “onePerCapacity”.

Here is a simple example of an executable application launching the "myCommand" command on the node provided by the "myNodeProvider" node provider:

```
<application>
  <executable>
    <command name="myCommand" />
    <nodeProvider refid="myNodeProvider" />
  </executable>
</application>
```

14.7.2.2. MPI

This type of application describes the deployment of an MPI application. It is a variation of the "Executable" type, but specifically designed for MPI applications. The chapter Chapter 9, *Wrapping Native MPI Application* shows three different usages of the MPI application type: to launch standalone MPI application, MPI applications coupling and Java-MPI coupling.

14.7.2.3. ProActive

This element describes the deployment of a ProActive-based application. It can have the following children:

- **<configuration>**: various configuration parameters
- **<technicalServices>**: set of technical services global to this instance of ProActive
- **<virtualNode>**: description of a virtual node. There can be any number of such an lement

And it has two mandatory attributes defining the ProActive home directory:

- **"relpath"**: relative path to the ProActive installation (relative to the base attribute)
- **"base"**: base path to the ProActive installation. It can be either "home" or "root". "home" represents the user’s home directory whereas "root" is the root directory of the system.

The **<configuration>** element can have the following child elements:

- **<bootClasspath>**: the boot classpath for the JVM
- **<java>**: the path to the Java executable
- **<jvmarg>**: arguments passed to the JVM
- **<applicationClasspath>**: classpath for the application
- **<proactiveClasspath>**: classpath used to override the standard ProActive classpath computed from its installation location
- **<securityPolicy>**: path to the Java security policy file
- **<proactiveSecurity>**: security policy for application and runtime. This element has two optional children:
  - **<applicationPolicy>**: path to Java security policy file that will be applied on the application's objects deployed at runtime, like nodes and active objects
  - **<runtimePolicy>**: path to Java security policy file that will be applied on the ProActive Runtime
- **<log4jProperties>**: path to the Java log4j configuration file
- **<userProperties>**: path to a property file (defining Java and/or ProActive properties)
- **<debug>**: to be completed

A technical service is a non-functional requirement that may be dynamically fulfilled at runtime by adapting the configuration of selected resources. For instance, we can use a technical service to activate the fault-tolerance mechanism on each node. For more information,

please refer to Chapter 20, *Technical Service*. A technical service can be defined at the application level, at the virtual node level or at the node provider level.

The **<virtualNode>** element can have the following children:

- **<technicalServices>**: a technical service specific to this virtual node. There can be any number of such children. Please refer to the previous paragraph.
- **<nodeProvider>**: node provider which will provide this virtual node with ProActive nodes. There can be as many node providers as you want. A nodeProvider element can have a **<technicalServices>** child defined a technical service specific to this node provider. A <nodeProvider> can also have the following attributes:
  - **"refid"** (mandatory): the id of the node provider (as defined in the resources element)
  - **"capacity"**: the capacity of this ProActive node provider (that is, the number of ProActive nodes which will be requested from it)

A **<virtualNode>** element can also have the following attributes:

- **"id"** (mandatory): a string identifying this virtual node. This identifier will be used in the Java program to deploy active objects on the nodes attached to this virtual node.
- **"capacity"** (optional): the capacity requested by this virtual node (that is, the total number of nodes it will request from the node providers which are affected to it). If no capacity is specified, then the virtual node will try to get as many nodes as possible. Such a virtual node is then characterized as greedy.

Here is an simple example of such a proactive element:

```
<application>
  <proactive base="root" relpath="myProActiveDirectory">
    <virtualNode id="myVirtualNode">
      <nodeProvider refid="myNodeProvider" />
    </virtualNode>
  </proactive>
</application>
```

Note

Your are not compelled to use virtual nodes. In that case, you have to handle your deployment using the three following methods of the org.objectweb.proactive.gcmdeployment.GCMApplication class:

- getAllNodes(): Returns all created or acquired Nodes
- getTopology(): Returns the topology of this GCM Application
- updateTopology(): Updates the Topology passed in parameter

Warning: **DO NOT USE THIS METHODS IF YOU HAVE DEFINED A VIRTUAL NODE**. If you do this, you will get an IllegalStateException.

14.7.3. The <resources> element

This element is used to define the node providers which are referenced in the <virtualNode> elements. The **<resources>** is therefore composed of a **"id"** attribute necessary to reference it. It is also consisted of a **<file>** child which informs the path to a GCM Deployment descriptor.

Here is a simple examples of such an element definition:

```
<resources>
  <nodeProvider id="myNodeProvider">
    <file path="GCMD.xml" />
  </nodeProvider>
</resources>
```

Chapter 15. XML Deployment Descriptors

15.1. Objectives

The main goal of using deployment descriptors is to remove all the deployment-related parameters from the source. Thus, a same source code can be used to deployed an application on different infrastructures.
Parameters tied to the deployment of an application should be totally described in a xml deployment descriptor. Hence within the source code, there are no longer any references to:

- **Machine names**
- **Creation Protocols**
  - local
  - ssh, gsissh, rsh, rlogin
  - lsf, pbs, sun grid engine, oar, prun
  - globus(GT2, GT3 and GT4), arc (nordugrid)
- **Registry/Lookup and Communications Protocols**
  - rmi
  - http
  - rmissh
  - ibis
  - soap
- **Files Transfers**
  - scp, rcp
  - arc (nordugrid)
  - other protocols like globus soon

A ProActive application can be deployed on different hosts, with different protocols **without** changing the source code

15.2. Principles

- **Within a ProActive program, active objects are created on Nodes as usual.**

```
PAActiveObject.newActive(className, constructorParameters, node);
```

- **Nodes can be obtained from VirtualNodes (VN) declared and defined in a ProActiveDescriptor**
- **Nodes are actual entities:**
  - running into a JVM, on a host
  - they are the results of mapping VN --> JVMs

  But it is the *VirtualNodes which are actually used in program source*.
- **After activation, the names of Nodes mapped to a VirtualNode are defined as the VirtualNode name concatenated to a random number.**
- **VNs have the following characteristics:**
  - a VN is uniquely identified with a String ID
  - a VN is defined in a ProActiveDescriptor
  - a VN has an object representation in a program after activation

- **A ProActiveDescriptor file specifies:**
  - the mapping between VNs and JVMs
  - the way to create, acquire JVMs using processes defined in the lower infrastructure part
  - local, remote processes or combination of both to create remote jvms.

    For instance defining an **sshProcess** that itself references a local **jvmProcess**. At execution, the ssh process will launch a jvm on the remote machine specified in hostname attribute of **sshProcess** definition.
  - files transfers
  - fault tolerance, security
- **Example:**

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

   <!-- Variable Definitions -->
   <variables>
      <descriptorVariable name="PROACTIVE_HOME" value="/user/ffonteno/home/proactive-git/
programming" />
      <descriptorVariable name="JAVA_HOME" value="/user/ffonteno/home/src/java/jdk" />
   </variables>

   <!-- Virtual Node Definitions -->
   <componentDefinition>
      <virtualNodesDefinition>
         <virtualNode name="VN1"/>
      </virtualNodesDefinition>
   </componentDefinition>

   <deployment>

      <!-- Mappings between Virtual Nodes and JVMs -->
      <mapping>
         <map virtualNode="VN1">
            <jvmSet>
               <vmName value="jvm" />
            </jvmSet>
         </map>
      </mapping>

      <!-- Mappings between JVMs and process references. -->
      <!-- Process references are used hereafter (within the infrastructure element)
          to describe the process used to create the JVMs. -->
      <jvms>
         <jvm name="jvm">
            <creation>
               <processReference refid="jvmProcess" />
            </creation>
         </jvm>
      </jvms>
```

```
  </deployment>
  <infrastructure>
    <processes>

      <!-- Process Definitions -->
      <processDefinition id="jvmProcess">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
        </jvmProcess>
      </processDefinition>

    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

This example shows a VirtualNode called **VN1**, which is mapped to a JVM called **jvm**.

This **jvm** will be created using the process called **jvmProcess** which is defined in the infrastructure part. This part will be discussed later on. But you can already notice that there are two parts in a descriptor file: an abstract one containing VirtualNode definitions and deployment information and a more concrete one containing concrete infrastructure information where all processes are defined.

- **Typical example of a program code:**

```
String descriptorFile = Main.class.getResource(
      "/org/objectweb/proactive/examples/documentation/XMLDeployment/SSHDescriptor.xml").getPath();

// Creates the ProActiveDescriptor corresponding to the descriptor file
ProActiveDescriptor proActiveDescriptor = PADeployment.getProactiveDescriptor(descriptorFile);

// Gets the virtual node named VN1 described in the descriptor file.
VirtualNode virtualNode = proActiveDescriptor.getVirtualNode("VN1");

// Activates the virtual node.
// For activating several virtual node at once, you can use
// proActiveDescriptorAgent.activateMappings()
virtualNode.activate();

String className = A.class.getName();
Object[] constructorParameters = new Object[] {};

// Gets a node on which the active object will be created
Node node = virtualNode.getNode();

// Creates the active object
A a = (A) PAActiveObject.newActive(className, constructorParameters, node);
```

org.objectweb.proactive.core.descriptor.data.ProActiveDescriptor and org.objectweb.proactive.core.descriptor.data.VirtualNode provides you with a set of methods to manipulate VirtualNodes. Please refer to the javadoc to learn more.

15.3. Different types of VirtualNodes

15.3.1. VirtualNodes Definition

- Mapping one to one: 1 VN --> 1 JVM

```
<virtualNodesDefinition>
  <virtualNode name='Dispatcher'/>
</virtualNodesDefinition>
<deployment>
  <mapping>
    <map virtualNode='Dispatcher'>
      <jvmSet>
        <vmName value='Jvm0'/>
      </jvmSet>
    </map>
  </mapping>
</deployment>
```

Another possibility for the one to one mapping is to map 1 VN to the jvm running the program. In that case the lookup protocol can be specified but is optional (default value is the property **proactive.communication.protocol**) as it is shown in the following:

```
<virtualNodesDefinition>
  <virtualNode name='Dispatcher'/>
</virtualNodesDefinition>
<deployment>
  <mapping>
    <map virtualNode='Dispatcher'>
      <jvmSet>
        <currentJVM protocol='rmi'/>
        <!-- or <currentJVM/> -->
      </jvmSet>
    </map>
  </mapping>
</deployment>
```

Since it is the current JVM, it does not have to be redefined later on in the descriptor. This will be shown in a complete example.

- Mapping one to n: 1 VN --> N JVMs

```
<virtualNodesDefinition>
  <virtualNode name='Renderer' property='multiple'/>
</virtualNodesDefinition>
<deployment>
  <mapping>
    <map virtualNode='Renderer'>
      <jvmSet>
        <currentJVM/>
        <vmName value='Jvm1'/>
        <vmName value='Jvm2'/>
        <vmName value='Jvm3'/>
        <vmName value='Jvm4'/>
      </jvmSet>
    </map>
  </mapping>
</deployment>
```

Note that the **property** attribute is set to **multiple** since you want to map 1 VN to multiple JVMs. Then a set of JVMs is defined for the VirtualNode **Renderer**.
Four values are possible for this **property** attribute:

- **unique** - one to one mapping
- **unique_singleAO** - one to one mapping and only one active object deployed on the corresponding node
- **multiple** - one to N mapping
- **multiple_cyclic** - one to N mapping in a cyclic manner.

This property is not mandatory but an exception can be thrown in case of incompatibility. For instance, an exception can be thrown if this property is set to unique and more than one jvm is defined in the jvmSet tag. In case of property set to **unique_singleAO**, the **getUniqueAO()** method of the org.objectweb.proactive.core.descriptor.data.VirtualNode class returns the unique AO created.

Three other attributes **timeout, waitForTimeout, minNodeNumber** can be set when defining a virtualNode

```
<virtualNodesDefinition>
  <virtualNode name='Dispatcher' timeout='200' waitForTimeout='true'/>
  <virtualNode name='Renderer' timeout='200' minNodeNumber='3'/>
</virtualNodesDefinition>
```

Depending on the value of **waitForTimeout**, the **timeout** attribute has two different meanings. If the **waitForTimeout** attribute, which is a boolean, is set to **true**, then you will have to wait exactly **timeout** milliseconds before accessing Nodes. If **waitForTimeout** is set to **false**, then **timeout** represents the maximum amount of time to wait, i.e. no more nodes will be created once this timeout over. Default value for **waitForTimeout** attribute is **false**.
The **minNodeNumber** attribute defines the minimum number of nodes to be created. If not defined, access to the nodes will occur once the timeout expires, or the number of nodes expected are effectively created. Setting this attribute allows to redefine the number of nodes expected, we define it as the number of nodes needed for the VirtualNode to be suitable for the application. In the example above, once **3** nodes are created and mapped to the VirtualNode **Renderer**, this VirtualNode starts to give access to its nodes. Those options are very useful when there is no idea about how many nodes will be mapped on the VirtualNode (which is often unusual). All those attributes are optional.

- Mapping n to one: N VN --> 1 JVMs

```
<virtualNodesDefinition>
  <virtualNode name='Dispatcher' property='unique_singleAO'/>
  <virtualNode name='Renderer' property='multiple'/>
</virtualNodesDefinition>
<deployment>
  <mapping>
    <map virtualNode='Dispatcher'>
      <jvmSet>
        <vmName value='Jvm1'/>
      </jvmSet>
    </map>
    <map virtualNode='Renderer'>
      <jvmSet>
        <vmName value='Jvm1'/>
        <vmName value='Jvm2'/>
        <vmName value='Jvm3'/>
        <vmName value='Jvm4'/>
      </jvmSet>
    </map>
  </mapping>
</deployment>
```

In this example both VirtualNodes **Dispatcher** and **Renderer** have a mapping with **Jvm1**, it means that at deployment time, both VirtualNodes will get nodes created in the same JVM. Here is the notion of **co-allocation** in a JVM.

- VirtualNode registration

Descriptors provide the ability to register a VirtualNode in a registry such as RMIRegistry, HTTP registry, IBIS/RMI Registry Service. Hence this VirtualNode will be accessible from another application as it is described in Section 15.3.2, “VirtualNodes Acquisition”. The protocol (registry) to use can be specified in the descriptor. If not specified, the VirtualNode will register using the protocol specified in the **proactive.communication.protocol** property which is **rmi** by default.

```
<virtualNodesDefinition>
   <virtualNode name='Dispatcher' property='unique_singleAO'/>
<virtualNodesDefinition/>
<deployment>
   <register virtualNode='Dispatcher' protocol='rmi'/>
   <!-- or <register virtualNode='Dispatcher'/> -->
   <!-- Using this syntax, registers the VirtualNode with the protocol
         specified in proactive.communication.protocol property -->
   <mapping>
      <map virtualNode='Dispatcher'>
         <jvmSet>
            <vmName value='Jvm0'/>
         </jvmSet>
      </map>
   </mapping>
</deployment>
```

The **register** tag allows to register the VirtualNode **Dispatcher** when activated, on the local machine in a RMIRegistry. As said before this VirtualNode will be accessible by another application using the lookup tag (see below) or using the lookupVirtualNode method of the PADeployment class.

15.3.2. VirtualNodes Acquisition

Descriptors provide the ability to acquire a VirtualNode already deployed by another application. Such VirtualNodes are defined in **VirtualNodes Acquisition** tag as it is done for **VirtualNodesDefinition** except that no property and no mapping with jvms are defined since such VNs are already deployed. In the deployment part, the lookup tag gives information on where and how to acquire the VirtualNode. Lookup will be performed when activating the VirtualNode.

```
<virtualNodesAcquisition>
   <virtualNode name='Dispatcher'/>
</virtualNodesAcquisition>
.....
<deployment>
   .....
   <lookup virtualNode='Dispatcher' host='machine_name' protocol='rmi' port='2020'/>
</deployment>
```

As mentioned in the previous section, in order to acquire VirtualNode **Dispatcher**, it must have previously been registered on the specified host by another application. Sometimes, the host where to perform the lookup will only be known at runtime. In that case, it is specified in the descriptor with '*' for the host attribute.

```
<lookup virtualNode='Dispatcher' host='*' protocol='rmi'/>
```

Then when the host name is available, ProActive provides the method **setRuntimeInformations** in the org.objectweb.proactive.core.descriptor.data.VirtualNode class to update the value and to perform the lookup. Typical example of code:

```
// Returns a ProActiveDescriptor object from the xml file
ProActiveDescriptor pad = PADeployment.getProactiveDescriptor(String xmlFileLocation);
```

```
// Activates all VirtualNodes (definition and acquisition)
pad.activateMappings();

// Gets the Dispatcher virtual node
VirtualNode vnDispatcher = pad.getVirtualNode('Dispatcher');

// Sets the host to lookup by setting the LOOKUP_HOST property
vnDispatcher.setRuntimeInformations('LOOKUP_HOST','machine_name);
```

To summarize, all VirtualNodes are activated when calling the activate method except if '*' is set for a VirtualNode to be acquired. In that case, the lookup will be performed when giving host information.

Registration and lookup can be performed automatically when using tags in the descriptor as well as programmatically using static methods provided by the org.objectweb.Proactive class:

```
PADeployment.registerVirtualNode(VirtualNode virtualNode, String registrationProtocol, boolean
 replacePreviousBinding);
```

```
PADeployment.lookupVirtualNode(String url, String protocol);
```

```
PADeployment.unregisterVirtualNode(VirtualNode virtualNode);
```

15.4. Different types of JVMs

15.4.1. Creation

- 1 JVM --> 1 Node

```
.....
<jvm name='jvm1'>
   <creation>
      <processReference refid='jvmProcess'/>
   </creation>
</jvm>
.....
```

In this example, jvm1 will be created using the process called **jvmProcess** (as discussed later on, this process represents a java process and can be seen as the java ProActiveClassname command)

- 1 JVM --> N Nodes on a single JVM

```
.....
<jvm name='jvm1' askedNodes='3'>
   <creation>
      <processReference refid='jvmProcess'/>
   </creation>
</jvm>
.....
```

- 1 JVM --> N Nodes on N JVMs

This is the case when the referenced process is a cluster process (LSF, PBS, GLOBUS, ...) or a process list (see Process list)

15.4.2. Acquisition

Descriptors give the ability to acquire JVMs instead of creating them. To do so, it must be specified in the **acquisition** tag which service to use in oder to acquire the JVMs. Services will be described below, in the infrastructure part. At this point one service is provided: the **RMIRegistryLookup** service.

```
<jvm name='jvm1'>
   <acquisition>
      <serviceReference refid='lookup'/>
   </acquisition>
</jvm>
.....
```

In this example, **Jvm1** will be acquired using the service called **lookup** (as discussed later on, this service represents a way to acquire a JVM). Note that the name **lookup** is totally arbitrary on condition that a service with the id **lookup** is defined in the infrastructure part.

15.5. Validation against XML Schema

To avoid mistake when building XML descriptors, ProActive provides an XML Schema called **DescriptorSchema.xsd**. To validate your file against this schema, the following line must be put at the top of the xml document as it is done for all ProActive examples:

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

Note that this schema is available in the ProActive distribution package in the PROACTIVE_HOME/src/Core/org/objectweb/proactive/core/descriptor/xml/schemas/deployment/3.3/deployment.xsd file. Using descriptors related methods (Proactive.getProactiveDescriptor(file)) triggers automatic and transparent validation of the file using Xerces2_4_0[1] if the ProActive property **schema.validation** is set to **enable** (see Chapter 11, *ProActive Basic Configuration* for more details). If a problem occurs during the validation, an error message is displayed. Otherwise, if the validation is successful, no message appears. An XML validation tool such as xmllint command on Unix or XMLSPY5.0 on Windows can also be used to validate XML descriptors.

15.6. Complete description and examples

The following XML files was used for the C3D application before being replaced by GCM Deployment descriptors. The first file is read when launching the C3DDispatcher. The second one is read every time a C3DUser is added. Both files contain many features described earlier in this document.

```
<ProActiveDescriptor
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xsi:noNamespaceSchemaLocation='DescriptorSchema.xsd'>
   <componentDefinition>
      <virtualNodesDefinition>
         <virtualNode name='Dispatcher' property='unique_singleAO'/>
         <virtualNode name='Renderer' property='multiple'/>
      </virtualNodesDefinition>
   </componentDefinition>
   <deployment>
      <register virtualNode='Dispatcher'/>
      <mapping>
         <map virtualNode='Dispatcher'>
            <jvmSet>
               <currentJvm/>
            </jvmSet>
         </map>
         <map virtualNode='Renderer'>
            <jvmSet>
               <vmName value='Jvm1'/>
```

[1] http://xml.apache.org/xerces2-j/index.html

```
            <vmName value='Jvm2'/>
            <vmName value='Jvm3'/>
            <vmName value='Jvm4'/>
          </jvmSet>
        </map>
      </mapping>
      <jvms>
        <jvm name='Jvm1'>
          <creation>
            <processReference refid='jvmProcess'/>
          </creation>
        </jvm>
        <jvm name='Jvm2'>
          <creation>
            <processReference refid='jvmProcess'/>
          </creation>
        </jvm>
        <jvm name='Jvm3'>
          <creation>
            <processReference refid='jvmProcess'/>
          </creation>
        </jvm>
        <jvm name='Jvm4'>
          <creation>
            <processReference refid='jvmProcess'/>
          </creation>
        </jvm>
      </jvms>
    </deployment>
    <infrastructure>
      <processes>
        <processDefinition id='jvmProcess'>
          <jvmProcess
            class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
        </processDefinition>
      </processes>
    </infrastructure>
</ProActiveDescriptor>
```

Example 15.1. C3D_Dispatcher_Render.xml

The abstract part containing VirtualNodes definition and deployment informations has already been explained. To summarize, two VirtualNodes are defined **Dispatcher** and **Renderer**. **Dispatcher** is mapped to the jvm running the main() method, and will be exported using the protocol specified in the **proactive.communication.protocol** property. This VirtualNode will be registered in a Registry (still using the protocol specified in **proactive.communication.protocol** property) when activated. As for **Renderer**, it is mapped to a set of JVMs called **Jvm1**, ..., **Jvm4**.

```
<ProActiveDescriptor
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='DescriptorSchema.xsd'>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name='User'/>
    </virtualNodesDefinition>
```

```
    <virtualNodesAcquisition>
      <virtualNode name='Dispatcher'/>
    </virtualNodesAcquisition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode='User'>
        <jvmSet>
          <currentJvm/>
        </jvmSet>
      </map>
    </mapping>
    <lookup virtualNode='Dispatcher' host='*' protocol='rmi'/>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id='jvmProcess'>
        <jvmProcess
          class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

Example 15.2. C3D_User.xml

This file is read when adding a C3DUser. Two VirtualNodes are defined: **User** which is mapped to the jvm running the main() method, whose acquisition method is performed by looking up the RMIRegistry, and **Dispatcher** in the **virtualNodesAcquisition** part which will be the result of a lookup in the RMIRegistry of a host to be specified at runtime.

15.7. Infrastructure and processes

In the previous example, all defined JVMs are created using **jvmProcess** process. This process is totally defined in the **infrastructure** part. Of course the process name in the creation part must point to an existing defined process in the **infrastructure** part. For instance, if the name in the creation tag is **localJVM**, there must be a process defined in the **infrastructure** with the id **localJVM**

15.7.1. Local JVMs

In the previous example, the defined process **jvmProcess** will create local JVMs. The class attribute defines the class to instantiate in order to create the process. ProActive library provides a class to instantiate in order to create processes that will launch local JVMs: org.objectweb.proactive.core.process.JVMNodeProcess

```
<infrastructure>
  <processes>
    <processDefinition id='jvmProcess'>
      <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcesss'>
        <classpath>
          <absolutePath value='/home/ProActive/classes'/>
          <absolutePath value='/home/ProActive/lib/bcel.jar'/>
          <absolutePath value='/home/ProActive/lib/asm.jar'/>
          <absolutePath value='/home/ProActive/lib/reggie.jar'/>
        </classpath>
        <javaPath>
          <absolutePath value='/usr/local/jdk1.4.0/bin/java'/>
```

```
        </javaPath>
        <policyFile>
          <absolutePath value='/home/ProActive/dist/proactive.java.policy'/>
        </policyFile>
        <log4jpropertiesFile>
          <relativePath origin='user.home' value='ProActive/dist/proactive-log4j'/>
        </log4jpropertiesFile>
        <ProActiveUserPropertiesFile>
          <absolutePath value='/home/config.xml'/>
        </ProActiveUserPropertiesFile>
        <jvmParameters>
          <parameter
            value='-Djava.library.path=/home1/fabrice/workProActive/ProActive/lib'/>
          <parameter
            value='-Dsun.boot.library.path=/home1/fabrice/workProActive/ProActive/lib'/>
          <parameter value='-Xms512 -Xmx512'/>
        </jvmParameters>
      </jvmProcess>
    </processDefinition>
  </processes>
</infrastructure>
```

As shown in the example above, **ProActive** provides the ability to define or change the **classpath** environment variable, the **java path**, the **policy file path**, the **log4j properties file path**, the **ProActive properties file path** (see Chapter 11, *ProActive Basic Configuration* for more details) and also to pass **parameters** to the JVM to be created.

Note

Note that parameters to be passed here are related to the jvm in opposition to properties given in the configuration file (see Chapter 11, *ProActive Basic Configuration*), which is more focused on ProActive or application behaviour. In fact parameters given here will be part of the java command to create other jvms, whereas properties given in the config file will be loaded once the jvm is created.

If not specified, there is a default value (except for the jvmParameters element) for each of these variables. In the first example of this section, only the **Id** of the process, and the **class** to instantiate are defined. If for example the home directory of the remote machine where you want to create a JVM is not the same as the one of your local machine, then you might want to define or redefine variable such as the **classpath** or **java path** or **policyfile path**. As shown in the example, **paths** to files can be either **absolute** or **relative**. If relative, an origin must be provided, it can be **user.home** or **user.dir** or **user.classpath** and it is resolved **locally**, i.e on the JVM reading the descriptor and not on the remote JVM that is going to be created.

As mentionned in the configuration file (see Chapter 11, *ProActive Basic Configuration*), if the <ProActiveUserPropertiesFile> is not defined for remote JVMs, they will load a default one once created.

Even if not shown in this example, a specific tag is provided for XbootClasspathi option under the form.

```
<bootclasspath>
  <relativePath origin='user.home' value='/IOFAb/Ibis/'/>
  <relativePath origin='user.home' value='/IOFAb/classlibs/jdk'/>
</bootclasspath>
```

15.7.2. Remote JVMs

With XML Deployment Descriptor, **ProActive** provides the ability to create remote Nodes (remote JVMs). You can specify in the descriptor if you want to access the remote host with **rsh**, **ssh**, **rlogin**, **lsf**, **pbs**, **oar**, **prun**, **globus**, **arc (nordugrid)**. How to use these

protocols is explained in the following examples. Just remind that you can also combine these protocols.The principle of combination is fairly simple, you can imagine for instance that you will log on a remote cluster frontend with **ssh**, then use **pbs** to book nodes and to create **JVMs** on each. You will also notice that there is at least one combination for each remote protocol. Indeed each remote protocol **must** have a pointer either on another remote protocol or on a **jvmProcess** to create a jvm (discussed previously).

You can find in the $PROACTIVE_HOME/descriptors/examples_legacy_descriptors/ directory several examples of supported protocols and useful combinations.

Note

For using the ProActive XML Deployment, ProActive has to be installed on the local host as well as on every machine you want to create Nodes on.

- RSH

```
.....
<jvm name='jvm1'>
  <creation>
    <processReference refid='rshProcess'/>
  </creation>
</jvm>
.....
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess
      class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='rshProcess'>
    <rshProcess
      class='org.objectweb.proactive.core.process.rsh.RSHProcess' hostname='sea.inria.fr'>
      <processReference refid='jvmProcess'/>
    </rshProcess>
  </processDefinition>
</processes>
```

For **jvm1** the creation process is **rshProcess** which is defined in the **infrastructure** section. To define this process you have to give the class to instantiate to create the **rsh** process. **ProActive** provides org.objectweb.proactive.core.process.rsh.RSHProcess to create **rsh** process. You must give the remote host name to log on with rsh. You can define as well username='toto' if you plan to use rsh with **-l option**. As said before this **rsh** process **must** reference a local process, and in the example, it references the process defined with the id **jvmProcess**. It means that once logged on sea.inria.fr with rsh, a local JVM will be launched, ie a ProActive node will be created on sea.inria.fr thanks to the process defined by **jvmProcess**.

Here is a complete RSH deployment example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor
  xmlns="urn:proactive:deployment:3.3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:proactive:deployment:3.3 http://www-sop.inria.fr/oasis/ProActive/schemas/
deployment/3.3/deployment.xsd">
  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/home/user/ProActive"/> <!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME" value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM ,
 CHANGE ME!!!! -->
  </variables>
```

```
<componentDefinition>
  <virtualNodesDefinition>
    <virtualNode name="matrixNode" property="multiple" />
  </virtualNodesDefinition>
</componentDefinition>
<deployment>
  <mapping>
    <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="localJVM" />
      </creation>
    </jvm>
    <jvm name="Jvm2">
      <creation>
        <processReference refid="rsh_crusoe" />
      </creation>
    </jvm>
    <jvm name="Jvm3">
      <creation>
        <processReference refid="rsh_waha" />
      </creation>
    </jvm>
    <jvm name="Jvm4">
      <creation>
        <processReference refid="rsh_amstel" />
      </creation>
    </jvm>
  </jvms>
</deployment>
<infrastructure>
  <processes>
    <processDefinition id="localJVM">
      <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
        <classpath>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
        </classpath>
        <javaPath>
          <absolutePath value="${JAVA_HOME}/bin/java"/>
```

```
            </javaPath>
            <policyFile>
              <absolutePath value="${PROACTIVE_HOME}/dist/proactive.java.policy"/>
            </policyFile>
            <log4jpropertiesFile>
              <absolutePath value="${PROACTIVE_HOME}/dist/proactive-log4j"/>
            </log4jpropertiesFile>
            <!--<jvmParameters>
              <parameter value="-Dproactive.communication.protocol=rmissh"/>
            </jvmParameters>-->
          </jvmProcess>
        </processDefinition>
        <processDefinition id="rsh_crusoe">
          <rshProcess
            class="org.objectweb.proactive.core.process.rsh.RSHProcess"
            hostname="crusoe.inria.fr">
            <processReference refid="localJVM"></processReference>
          </rshProcess>
        </processDefinition>
        <processDefinition id="rsh_waha">
          <rshProcess
            class="org.objectweb.proactive.core.process.rsh.RSHProcess"
            hostname="waha.inria.fr">
            <processReference refid="localJVM"></processReference>
          </rshProcess>
        </processDefinition>
        <processDefinition id="rsh_amstel">
          <rshProcess
            class="org.objectweb.proactive.core.process.rsh.RSHProcess"
            hostname="amstel.inria.fr">
            <processReference refid="localJVM"></processReference>
          </rshProcess>
        </processDefinition>
      </processes>
    </infrastructure>
</ProActiveDescriptor>
```

- RLOGIN

```
.....
<jvm name='jvm1'>
  <creation>
    <processReference refid='rloginProcess'/>
  </creation>
</jvm>
.....
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess
      class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='rloginProcess'>
    <rloginProcess
      class='org.objectweb.proactive.core.process.rlogin.RLoginProcess'
      hostname='sea.inria.fr'>
```

```
        <processReference refid='jvmProcess'/>
      </rloginProcess>
    </processDefinition>
</processes>
```

You can use **rlogin** in the same way that you would use **rsh**

- SSH

```
.....
<jvm name='jvm1'>
    <creation>
      <processReference refid='sshProcess'/>
    </creation>
</jvm>
.....
<processes>
    <processDefinition id='jvmProcess'>
      <jvmProcess
        class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
    </processDefinition>
    <processDefinition id='sshProcess'>
      <sshProcess
        class='org.objectweb.proactive.core.process.ssh.SSHProcess'
        hostname='sea.inria.fr'>
        <processReference refid='jvmProcess'/>
      </sshProcess>
    </processDefinition>
</processes>
```

ProActive provides org.objectweb.proactive.core.process.ssh.SSHProcess to create **ssh** process.

In order to use ssh to log on a remote host, you must perform some actions. First you need to copy your public key (located in identity.pub under ~/.ssh on your local machine) in the authorized_keys file (located under ~/.ssh) of the remote host. Then to avoid interactivity, you will have to launch on the local host the ssh-agent command: **ssh-agent $SHELL**. This command can be put in your .xsession file, in order to run it automatically when logging on your station. Then launching **ssh-add** command to add your identity, you will be asked to enter your **passphrase**, the one you provided when you have generated your ssh key pair.

Note that if the generated key pair is not encrypted (no passphrase), you do not need to run neither the ssh-agent, nor the ssh-add command. Indeed it is sufficient when using non encrypted private key, to only copy the public key on the remote host (as mentionned above) in order to get logged automatically on the remote host.

These steps must be performed **before** running any ProActive application using **ssh** protocol. If you are not familiar with ssh, see openSSH[2]

The following is a complete SSH deployment example.

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

[2] http://www.openssh.org

```
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="matrixNode" property="multiple" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="matrixNode">
        <jvmSet>
          <vmName value="Jvm1" />
          <vmName value="Jvm2" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="ssh_crusoe" />
        </creation>
      </jvm>
      <jvm name="Jvm2">
        <creation>
          <processReference refid="ssh_waha" />
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess" />
      </processDefinition>
      <processDefinition id="ssh_crusoe">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="crusoe.inria.fr">
          <processReference refid="localJVM"></processReference>
        </sshProcess>
      </processDefinition>
      <processDefinition id="ssh_waha">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="waha.inria.fr">
          <processReference refid="localJVM"></processReference>
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- Process List

ProActive provides a way to define a list of processes for **RSH, SSH, RLOGIN** protocols. Using **processList** or **processListbyHost** elements avoids having a long deployment file when many machines with similar names are going to be connected with protocols mentionned before. The first example below shows how to use **processList** tag and the second one how to use **processListbyHost**.

```
.....
<jvm name='jvm1'>
  <creation>
    <processReference refid='processlist'/>
  </creation>
</jvm>
.....
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess
      class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='processlist'>
    <processList
      class='org.objectweb.proactive.core.process.ssh.SSHProcessList'
      fixedName='node-' list='[0-100;2]^[10,20]'
      padding='3' domain='sophia.grid5000.fr'>
      <processReference refid='jvmProcess'/>
    </processList>
  </processDefinition>
</processes>
```

When using **processList tag**, the **class** attribute can take 3 values:

- org.objectweb.proactive.core.process.ssh.SSHProcessList

```
public class SSHProcessList extends AbstractListProcessDecorator {

  /**
   *
   */
  public SSHProcessList() {
    super();
  }

  /**
   * @see org.objectweb.proactive.core.process.AbstractListProcessDecorator#createProcess()
   */
  @Override
  protected ExternalProcessDecorator createProcess() {
    return new SSHProcess();
  }
}
```

- org.objectweb.proactive.core.process.rsh.RSHProcessList

```
public class RSHProcessList extends AbstractListProcessDecorator {

  /**
   *
   */
```

```
    public RSHProcessList() {
        super();
    }

    /**
     * @see org.objectweb.proactive.core.process.AbstractListProcessDecorator#createProcess()
     */
    @Override
    protected ExternalProcessDecorator createProcess() {
        return new RSHProcess();
    }
}
```

- org.objectweb.proactive.core.process.rlogin.RLoginProcessList

```
public class RLoginProcessList extends AbstractListProcessDecorator {

    /**
     *
     */
    public RLoginProcessList() {
        super();
    }

    /**
     * @see org.objectweb.proactive.core.process.AbstractListProcessDecorator#createProcess()
     */
    @Override
    protected ExternalProcessDecorator createProcess() {
        return new RLoginProcess();
    }
}
```

The **fixedName** attribute is mandatory and represents the fixed part shared by all machine names.
The **list** attribute is also mandatory and can take several forms:

- **[m-n]** means from m to n with a step 1,
- **[m-n;k]** means from m to n with a step k (m, m+k, m+2k, ...),
- **[m-n]^[x,y]** means from m to n excluding x and y,
- **[m-n]^[x,y-z]** means from m to n excluding x and values from y to z,
- **[m-n;k]^[x,y]** same as before except that the step is k.

The **padding** attribute is optional (default is 1) and represents the number of digits to use for the node number. For instance, the node 1 with a padding equal to 3 will be written 001.
Finally, the **domain** attribute is mandatory and represents the last part shared by all machine names. Thus, in the previous example, a jvm is going to be created using ssh on machines: node000.sophia.grid5000.fr, node002.sophia.grid5000.fr,..., node098.sophia.grid5000.fr, node100.sophia.grid5000.fr (note that step is 2) excluding machines: node010.sophia.grid5000.fr and node020.sophia.grid5000.fr.

```
.....
<jvm name='jvm1'>
    <creation>
        <processReference refid='processlist'/>
    </creation>
</jvm>
.....
```

```
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess
      class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='processlist'>
    <processListbyHost
      class='org.objectweb.proactive.core.process.ssh.SSHProcessList'
      hostlist='crusoe waha nahuel' domain='inria.fr'>
      <processReference refid='jvmProcess'/>
    </processListbyHost>
  </processDefinition>
</processes>
```

Using **processListbyHost** element allows to give a hostlist separated with a whitespace. The class attribute is defined as described in the processList tag. The **domain** attribute is optional since the complete hostname can also be provided in the hostlist attribute. In the example, a jvm is going to be created using ssh on crusoe.inria.fr, waha.inria.fr, nahuel.inria.fr.

Here are complete examples of SSH deployment using processList and processListbyHost.

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/home/user/ProActive"/> <!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME" value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM ,
 CHANGE ME!!!! -->
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="plugtest"/>
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="plugtest">
        <jvmSet>
          <vmName value="Jvm1"/>
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="ssh_list"/>
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
```

```
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
          </classpath>
          <javaPath>
            <absolutePath value="${JAVA_HOME}/bin/java"/>
          </javaPath>
          <policyFile>
            <absolutePath value="${PROACTIVE_HOME}/dist/proactive.java.policy"/>
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath value="${PROACTIVE_HOME}/dist/proactive-log4j"/>
          </log4jpropertiesFile>
          <!--<jvmParameters>
            <parameter value="-Dproactive.communication.protocol=rmissh"/>
          </jvmParameters>-->
        </jvmProcess>
      </processDefinition>
      <processDefinition id="ssh_list">
        <processList class="org.objectweb.proactive.core.process.ssh.SSHProcessList"
 fixedName="125.110.118." list="[96-200]^[96,102,103,104,105,110,11,112]" domain="" username="rquilici">
 <!--CHANGE ME!!!! -->
          <processReference refid="localJVM"></processReference>
        </processList>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

and

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

  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="matrixNode" property="multiple" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="matrixNode">
```

```
            <jvmSet>
              <vmName value="Jvm1" />
            </jvmSet>
          </map>
        </mapping>
        <jvms>
          <jvm name="Jvm1">
            <creation>
              <processReference refid="ssh_list" />
            </creation>
          </jvm>
        </jvms>
      </deployment>
      <infrastructure>
        <processes>
          <processDefinition id="localJVM">
            <jvmProcess
              class="org.objectweb.proactive.core.process.JVMNodeProcess" />
          </processDefinition>
          <processDefinition id="ssh_list">
            <processListbyHost
              class="org.objectweb.proactive.core.process.ssh.SSHProcessList"
              hostlist="crusoe waha amstel"> <!--CHANGE ME!!!! -->
              <processReference refid="localJVM"></processReference>
            </processListbyHost>
          </processDefinition>
        </processes>
      </infrastructure>
    </ProActiveDescriptor>
```

- LSF

  This protocol is used to create Nodes (JVMs) on a cluster. **ProActive** provides org.objectweb.proactive.core.process.lsf.LSFBSubProcess to create **bsub** process.

  In this part, we assume that you want to submit a job from a machine which is not the cluster frontend. As described before, you can combine protocols. In this case , you will have to define a process to log on the front-end of the cluster (**rlogin** if your machine is on the same LAN than the cluster front-end, else **ssh** (Remember that to use **ssh** you will have to run some commands as explained above).

```
<jvm name='Jvm2'>
  <creation>
    <processReference refid='sshProcess'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='bsubInriaCluster'>
    <bsubProcess
      class='org.objectweb.proactive.core.process.lsf.LSFBSubProcess'>
        <processReference refid='jvmProcess'/>
        <bsubOption>
```

```
            <hostlist>cluster_machine1 cluster_machine2<hostlist/>
            <processor>6</processor>
            <scriptPath>
              <absolutePath
                value='/home/ProActive/dist/scripts/unix/cluster/startRuntime.sh'/>
            </scriptPath>
          </bsubOption>
      </bsubProcess>
    </processDefinition>
    <processDefinition id='sshProcess'>
      <sshProcess
        class='org.objectweb.proactive.core.process.ssh.SSHProcess'
        hostname='sea.inria.fr'>
        <processReference refid='bsubInriaCluster'/>
      </sshProcess>
    </processDefinition>
</processes>
```

In this example, the JVM called **Jvm2** will be created using **ssh** to log on the cluster front end. Then a **bsub** command will be generated thanks to the process defined by **bsubInriaCluster**. This **bsub** command will create Nodes on several cluster machines, since **bsubInriaCluster** references the **jvmProcess** defined process. All tags defined under **<bsubOption>** are not mandatory, but they can be very useful. The <hostlist> tag defines possible candidates in the job attribution, if not set the job will be allocated among all cluster's machines. The **<processor>** tag defines the number of processor requested, if not set, one processor is requested. The **<resourceRequirement>** tag defines the expected number of processors per machine. For instance **<resourceRequirement value='span[ptile=2]'/>** ensures that 2 processors per machines will be used, whereas **<resourceRequirement value='span[ptile=1]'/>** forces LSF to allocate only one processor per machine. It represents the **-R** option of LSF. At last, **<scriptPath>** defines the path on the cluster front end of the **startRuntime.sh** script which is necessary to run ProActive on a cluster. This script is located under the Proactive/dist/scripts/unix/cluster directory. If not set the default location is set as ~/Proactive/dist/scripts/unix/cluster.

If you want to submit the job directly from the cluster entry point, define only the **bsubProcess** like in the previous example and skip the ssh definition.

```
<jvm name='Jvm2'>
  <creation>
    <processReference refid='bsubInriaCluster'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='bsubInriaCluster'>
    <bsubProcess
      class='org.objectweb.proactive.core.process.lsf.LSFBSubProcess'
      interactive='true' queue='short'>
      <processReference refid='jvmProcess'/>
      <bsubOption>
        <hostlist>cluster_machine1 cluster_machine2<hostlist/>
        <processor>6</processor>
        <scriptPath>
          <absolutePath value='/home/ProActive/dist/scripts/unix/cluster/startRuntime.sh'/>
```

```
            </scriptPath>
          </bsubOption>
        </bsubProcess>
      </processDefinition>
    </processes>
```

Note that in the example above two new attributes has appeared: **interactive** and **queue**. They are optional, and have a default value: respectively **false** and **normal**. They represent option in the bsub command: interactive mode, and the name of the queue.

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME"
      value="/home/user/ProActive" /><!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME"
      value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="plugtest" timeout="160000" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="plugtest">
        <jvmSet>
          <vmName value="Jvm1" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="sshInriaCluster" />
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM1">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
```

```
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
          </classpath>
          <javaPath>
            <absolutePath
              value="${JAVA_HOME}/bin/java" />
          </javaPath>
          <policyFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive.java.policy" />
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive-log4j" />
          </log4jpropertiesFile>
          <jvmParameters>
            <parameter
              value="-Dproactive.communication.protocol=rmissh" />
          </jvmParameters>
        </jvmProcess>
      </processDefinition>
      <processDefinition id="bsubInriaCluster">
        <bsubProcess
          class="org.objectweb.proactive.core.process.lsf.LSFBSubProcess">
          <processReference refid="localJVM1" />
          <bsubOption>
            <processor>60</processor>
            <resourceRequirement value="span[ptile=2]" />
            <scriptPath>
              <absolutePath
                value="${PROACTIVE_HOME}/scripts/unix/cluster/startRuntime.sh" />
            </scriptPath>
          </bsubOption>
        </bsubProcess>
      </processDefinition>
      <processDefinition id="sshInriaCluster">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="frontend" username="plugtest">
          <processReference refid="bsubInriaCluster" />
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- PBS

  This protocol is used to create jobs on cluster managed by PBS, PBSPro or Torque. ProActive provides org.objectweb.proactive.core.process.pbs.PBSBSubProcess to create **pbs** processes. As explained for LSF, you can combine protocols in order to log on the cluster's frontal with ssh and then to create nodes using PBS, or you can also use only PBS without ssh if you are already logged on the frontend. Example below shows how to combine an ssh process to log on the cluster and a PBS process that references a **jvmProcess** in order to create nodes on processors requested by PBS.

```
<jvm name='Jvm2'>
```

```
  <creation>
    <processReference refid='sshProcess'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='pbsCluster'>
    <pbsProcess class='org.objectweb.proactive.core.process.pbs.PBSSubProcess'>
      <processReference refid='jvmProcess'/>
      <pbsOption>
        <hostsNumber>4</hostsNumber>
        <processorPerNode>1</processorPerNode>
        <bookingDuration>00:15:00</bookingDuration>
        <outputFile>/home1/rquilici/out.log</outputFile>
        <scriptPath>
          <absolutePath value='/home/ProActive/dist/scripts/unix/cluster/pbsStartRuntime.sh'/>
        </scriptPath>
      </pbsOption>
    </pbsProcess>
  </processDefinition>
  <processDefinition id='sshProcess'>
    <sshProcess
      class='org.objectweb.proactive.core.process.ssh.SSHProcess'
      hostname='frontend'>
      <processReference refid='pbsCluster'/>
    </sshProcess>
  </processDefinition>
</processes>
```

Note that not all options are listed here, and some options mentionned in the example are optional:

- **hostsNumber** represents the number of host requested using pbs (default is 1)
- **processorPerNode** represents the number of processor per hosts requested (1 or 2, default is 1)
- **bookingDuration** represents the duration of the job (default is 1 minute)
- **outputFile** represents the file where to put the ouput of the job(default is specified by pbs)
- **scriptPath** represents the location on the frontend_host of the script pbsStartRuntime.sh (default is /user.home/ProActive/dist/scripts/unix/cluster/pbsStartRuntime.sh)

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME"
      value="/home/user/ProActive" /><!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME"
      value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
  </variables>
```

```
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="plugtest" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="plugtest">
        <jvmSet>
          <vmName value="Jvm1" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="sshInriaCluster" />
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM1">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
                                        <absolutePath value="${PROACTIVE_HOME}/dist/lib/
ProActive.jar"/>
                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>

                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
                          <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>

          </classpath>
          <javaPath>
            <absolutePath
              value="${JAVA_HOME}/bin/java" />
          </javaPath>
          <policyFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive.java.policy" />
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive-log4j" />
          </log4jpropertiesFile>
          <jvmParameters>
            <parameter
              value="-Dproactive.communication.protocol=rmissh" />
          </jvmParameters>
```

```
        </jvmProcess>
      </processDefinition>
      <processDefinition id="pbsInriaCluster">
        <pbsProcess
          class="org.objectweb.proactive.core.process.pbs.PBSSubProcess">
          <processReference refid="localJVM1" />
          <commandPath value="/opt/torque/bin/qsub" />
          <pbsOption>
            <hostsNumber>32</hostsNumber>
            <processorPerNode>2</processorPerNode>
            <bookingDuration>02:00:00</bookingDuration>
            <scriptPath>
              <!--absolutePath value="/home/plugtest/ProActive/scripts/unix/cluster/pbsStartRuntime.sh"/-->
              <absolutePath
                value="${PROACTIVE_HOME}/scripts/unix/cluster/pbsStartRuntime.sh" />
            </scriptPath>
          </pbsOption>
        </pbsProcess>
      </processDefinition>
      <processDefinition id="sshInriaCluster">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="frontend" username="plugtest">
          <processReference refid="pbsInriaCluster" />
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- Sun Grid Engine

  This protocol is used to create jobs on cluster managed by Sun Grid Engine. ProActive provides org.objectweb.proactive.core.process.gridengine.GridEngineSubProcess to create **grid engine** processes. As explained above, you can combine protocols in order to log on the cluster's frontal with ssh and then to create nodes using SGE, or you can also use only SGE without ssh if you are already logged on the frontend. The example below shows how to combine an ssh process to log on the cluster and a SGE process that references a **jvmProcess** in order to create nodes on processors requested by SGE.

```
<jvm name='Jvm2'>
  <creation>
    <processReference refid='sshProcess'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='sgeCluster'>
    <gridengineProcess class='org.objectweb.proactive.core.process.gridengine.GridEngineSubProcess'>
      <processReference refid='jvmProcess'/>
      <gridEngineOption>
        <hostsNumber>4</hostsNumber>
        <bookingDuration>00:15:00</bookingDuration>
        <scriptPath>
```

```
            <absolutePath value='/home/ProActive/dist/scripts/unix/cluster/gridEngineStartRuntime.sh'/>
          </scriptPath>
          <parallelEnvironment>mpi</parallelEnvironment>
        </gridEngineOption>
      </gridengineProcess>
    </processDefinition>
    <processDefinition id='sshProcess'>
      <sshProcess
        class='org.objectweb.proactive.core.process.ssh.SSHProcess'
        hostname='frontend'>
        <processReference refid='sgeCluster'/>
      </sshProcess>
    </processDefinition>
</processes>
```

As mentionned previously, many options exist, and correspond to the main options specified in an SGE system. For example, **ScriptPath** represents the location on the frontend_host of the script gridEngineStartRuntime.sh (default is /user.home/ProActive/dist/scripts/unix/cluster/gridEngineStartRuntime.sh).

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

    <variables>
      <descriptorVariable name="PROACTIVE_HOME"
        value="/home/user/ProActive" /><!--CHANGE ME!!!! -->
    </variables>
    <componentDefinition>
      <virtualNodesDefinition>
        <virtualNode name="plugtest" />
      </virtualNodesDefinition>
    </componentDefinition>
    <deployment>
      <mapping>
        <map virtualNode="plugtest">
          <jvmSet>
            <vmName value="Jvm1" />
          </jvmSet>
        </map>
      </mapping>
      <jvms>
        <jvm name="Jvm1">
          <creation>
            <processReference refid="sshCluster" />
          </creation>
        </jvm>
      </jvms>
    </deployment>
    <infrastructure>
      <processes>
        <processDefinition id="internalJVM">
```

```
    <jvmProcess
      class="org.objectweb.proactive.core.process.JVMNodeProcess">
      <classpath>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
        <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
      </classpath>
      <javaPath>
        <absolutePath
          value="/home/plugtest/j2sdk1.4.2_05/bin/java" /> <!--CHANGE ME!!!! -->
      </javaPath>
      <policyFile>
        <absolutePath
          value="${PROACTIVE_HOME}/dist/proactive.java.policy" />
      </policyFile>
      <log4jpropertiesFile>
        <absolutePath
          value="${PROACTIVE_HOME}/dist/proactive-log4j" />
      </log4jpropertiesFile>
      <jvmParameters>
        <parameter
          value="-Dproactive.communication.protocol=rmissh" />
      </jvmParameters>
    </jvmProcess>
  </processDefinition>
  <processDefinition id="sgeprocess">
    <gridEngineProcess
      class="org.objectweb.proactive.core.process.gridengine.GridEngineSubProcess"
      queue="normal">
      <processReference refid="internalJVM" />
      <commandPath
        value="/opt/gridengine/bin/lx26-x86/qsub" />
      <gridEngineOption>
        <hostsNumber>10</hostsNumber>
        <bookingDuration>3600</bookingDuration>
        <scriptPath>
          <absolutePath
            value="${PROACTIVE_HOME}/scripts/unix/cluster/gridEngineStartRuntime.sh" />
        </scriptPath>
        <parallelEnvironment>mpi</parallelEnvironment>
      </gridEngineOption>
    </gridEngineProcess>
  </processDefinition>
  <processDefinition id="sshCluster">
    <sshProcess
      class="org.objectweb.proactive.core.process.ssh.SSHProcess"
      hostname="frontend" username="plugtest"> <!--CHANGE ME!!!! -->
      <processReference refid="sgeprocess" />
    </sshProcess>
  </processDefinition>
```

```
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- OAR

OAR is a cluster protocol developed at INRIA Alpes and used on Grid5000[3]. ProActive provides org.objectweb.proactive.core.process.oar.OARSubProcess to use such a protocol. As explained above, you can combine protocols in order to log on the cluster's frontend with ssh and then to create nodes using OAR, or you can also use only OAR without ssh if you are already logged on the frontend. The example below shows how to combine an ssh process to log on the cluster, then an OAR process that references a **jvmProcess** in order to create nodes on processors requested by OAR.

```
<jvm name='Jvm2'>
  <creation>
    <processReference refid='sshProcess'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
  </processDefinition>
  <processDefinition id='oarCluster'>
    <oarProcess class='org.objectweb.proactive.core.process.oar.OARSubProcess'>
      <processReference refid='jvmProcess'/>
      <oarOption>
        <resources>node=2,weight=2</resources>
        <scriptPath>
          <absolutePath value='/home/ProActive/dist/scripts/unix/cluster/oarStartRuntime.sh'/>
        </scriptPath>
      </oarOption>
    </oarProcess>
  </processDefinition>
  <processDefinition id='sshProcess'>
    <sshProcess
      class='org.objectweb.proactive.core.process.ssh.SSHProcess'
      hostname='frontend'>
      <processReference refid='oarCluster'/>
    </sshProcess>
  </processDefinition>
</processes>
```

As mentionned previously, many options exist, and correspond to the main options specified in an OAR system. For example, **ScriptPath** represents the location on the frontend host of the script oarStartRuntime.sh (default is /user.home/ProActive/dist/scripts/unix/cluster/oarStartRuntime.sh).

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

[3] http://www.grid5000.fr

```
<variables>
  <descriptorVariable name="PROACTIVE_HOME"
    value="/home/user/ProActive" /><!--CHANGE ME!!!! -->
  <descriptorVariable name="JAVA_HOME"
    value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
</variables>
<componentDefinition>
  <virtualNodesDefinition>
    <virtualNode name="Test" property="multiple" />
  </virtualNodesDefinition>
</componentDefinition>
<deployment>
  <mapping>
    <map virtualNode="Test">
      <jvmSet>
        <vmName value="JvmSSH" />
      </jvmSet>
    </map>
  </mapping>
  <jvms>
    <jvm name="JvmOARGrid">
      <creation>
        <processReference refid="oarGridProcess" />
      </creation>
    </jvm>
    <jvm name="JvmSSH">
      <creation>
        <processReference refid="sshProcess" />
      </creation>
    </jvm>
  </jvms>
</deployment>
<infrastructure>
  <processes>
    <processDefinition id="jvmProcess">
      <jvmProcess
        class="org.objectweb.proactive.core.process.JVMNodeProcess">
        <classpath>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
          <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
        </classpath>
        <javaPath>
          <absolutePath
            value="${JAVA_HOME}/bin/java" />
        </javaPath>
        <policyFile>
          <absolutePath
            value="${PROACTIVE_HOME}/dist/proactive.java.policy" />
        </policyFile>
```

```
            <log4jpropertiesFile>
              <absolutePath
                value="${PROACTIVE_HOME}/dist/proactive-log4j" />
            </log4jpropertiesFile>
          </jvmProcess>
        </processDefinition>
        <processDefinition id="oarGridProcess">
          <oarGridProcess
            class="org.objectweb.proactive.core.process.oar.OARGRIDSubProcess"
            bookedNodesAccess="ssh" queue="default">
            <processReference refid="jvmProcess" />
            <commandPath value="/usr/local/bin/oargridsub" />
            <oarGridOption>
              <!--Available clusters are:
                | idpot       | caddo.imag.fr                |
                | gdx         | devgdx002.orsay.grid5000.fr    |
                | toulouse    | oar.toulouse.grid5000.fr       |
                | sophia      | oar.sophia.grid5000.fr         |
                | lyon        | oar.lyon.grid5000.fr           |
                | parasol     | oar.rennes.grid5000.fr         |
                | tartopom    | dev-powerpc.rennes.grid5000.fr |
                | paraci      | dev-xeon.rennes.grid5000.fr    |
                | icluster2   | ita101.imag.fr                 |
              -->
              <resources>
                sophia:nodes=2,lyon:nodes=1
              </resources>
              <walltime>00:03:00</walltime><!-- hour:min:sec-->
              <scriptPath>
                <!--relativePath origin="user.home" value="Proactive/scripts/unix/cluster/
oarGridStartRuntime.sh"/-->
                <absolutePath
                  value="${PROACTIVE_HOME}/scripts/unix/cluster/oarGridStartRuntime.sh" />
              </scriptPath>
            </oarGridOption>
          </oarGridProcess>
        </processDefinition>

        <processDefinition id="sshProcess">
          <sshProcess
            class="org.objectweb.proactive.core.process.ssh.SSHProcess"
            hostname="oar.grenoble.grid5000.fr">
            <processReference refid="oarGridProcess" />
          </sshProcess>
        </processDefinition>

      </processes>
    </infrastructure>
</ProActiveDescriptor>
```

- PRUN

PRUN is a cluster protocol developed at Amsterdam to manage their cluster[4]. ProActive provides org.objectweb.proactive.core.process.prun.PrunSubProcess to use such a protocol.

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


   <variables>
      <descriptorVariable name="PROACTIVE_HOME" value="/home/user/ProActive"/> <!--CHANGE ME!!!! -->
      <descriptorVariable name="JAVA_HOME"
         value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
   </variables>
   <componentDefinition>
      <virtualNodesDefinition>
         <virtualNode name="plugtest" timeout="120000"/>
      </virtualNodesDefinition>
   </componentDefinition>
   <deployment>
      <mapping>
         <map virtualNode="plugtest">
            <jvmSet>
               <vmName value="Jvm1"/>
            </jvmSet>
         </map>
      </mapping>
      <jvms>
         <jvm name="Jvm1">
            <creation>
               <processReference refid="sshProcess"/>
            </creation>
         </jvm>
      </jvms>
   </deployment>
   <infrastructure>
      <processes>
         <processDefinition id="linuxJVM1">
            <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
               <classpath>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
                  <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
               </classpath>
               <javaPath>
```

[4] http://www.cs.vu.nl/das/prun/prun.1.html

```
            <absolutePath value="${JAVA_HOME}/bin/java"/>
          </javaPath>
          <policyFile>
            <absolutePath value="${PROACTIVE_HOME}/dist/proactive.java.policy"/>
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath value="${PROACTIVE_HOME}/dist/proactive-log4j"/>
          </log4jpropertiesFile>
        </jvmProcess>
      </processDefinition>
      <processDefinition id="prunCluster">

<prunProcess class="org.objectweb.proactive.core.process.prun.PrunSubProcess" queue="plugtest">
          <processReference refid="linuxJVM1"/>
          <commandPath value="/usr/local/VU/reserve.sge/bin/prun"/>
          <prunOption>
            <hostsNumber>20</hostsNumber>
            <processorPerNode>2</processorPerNode>
            <bookingDuration>02:00:00</bookingDuration>
          </prunOption>
        </prunProcess>
      </processDefinition>
      <processDefinition id="sshProcess">

<sshProcess class="org.objectweb.proactive.core.process.ssh.SSHProcess" hostname="frontend" username="rqui
<!--CHANGE ME!!!! -->
          <processReference refid="prunCluster"/>
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- GLOBUS

  Like **ssh**, using **Globus** requires some steps to be performed. In particular the **java COG Kit** (no need for the whole GT) must be installed on the machine that will originates the **RSL** request. See COG Kit Installation[5] to know how to install the client kit. Then you have to initialize your proxy by running **COG_INSTALLATION/bin /grid-proxy-init**. You will be asked for a passphrase, which is the one you provided when requesting a user certificate at globus.org. Once these steps are performed, you can run **ProActive** application using **GRAM** protocol.

  ProActive provides org.objectweb.proactive.core.process.globus.GlobusProcess to create **globus** process.

```
<jvm name='Jvm2'>
  <creation>
    <processReference refid='globusProcess'/>
  </creation>
</jvm>
.....................................................
<processes>
  <processDefinition id='jvmProcess'>
    <jvmProcess class='org.objectweb.proactive.core.process.JVMNodeProcess'/>
```

[5] http://www.cogkit.org/

```
  </processDefinition>
  <processDefinition id='globusProcess'>
    <globusProcess
      class='org.objectweb.proactive.core.process.globus.GlobusProcess'
      hostname='globus1.inria.fr'>
      <processReference refid='jvmProcess'/>
      <environment>
        <variable name='DISPLAY' value='machine_name0.0'/>
      </environment>
      <globusOption>
        <count>10</count>
      </globusOption>
    </globusProcess>
  </processDefinition>
</processes>
```

In this example, **Jvm2** will be created using **GRAM**. An **RSL** request will be generated with information provided in the descriptor. For instance, the <environment> tag is not mandatory, but for the globus host to export the DISPLAY on your machine, you can define the value in the descriptor as well as other environment variable, except the classpath (or java path,...) which must be defined in the **jvmProcess** referenced by **globusProcess** as explained before. <globusOption> is not mandatory either. Default value for <count> element is 1. It represents the number of requested processor.

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME"
      value="/home/user/ProActive" /><!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME"
    value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
    <descriptorVariable name="GLOBUS_USER_HOME"
      value="/globus/home/user"
    /> <!--CHANGE ME!!!! -->
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="plugtest" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="plugtest">
        <jvmSet>
          <vmName value="Jvm1" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
```

```
            <processReference refid="globusProcess" />
          </creation>
        </jvm>

    </jvms>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM1">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
          </classpath>
          <javaPath>
            <absolutePath
              value="${JAVA_HOME}/bin/java" />
          </javaPath>
          <policyFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive.java.policy" />
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath
              value="${PROACTIVE_HOME}/dist/proactive-log4j" />
          </log4jpropertiesFile>
          <jvmParameters>
            <parameter
              value="-Dproactive.communication.protocol=http" />
            <parameter value="-Dproactive.http.port=22500" />
          </jvmParameters>
        </jvmProcess>
      </processDefinition>
      <processDefinition id="globusProcess">
        <globusProcess
          class="org.objectweb.proactive.core.process.globus.GlobusProcess"
          hostname="globus_frontend">
          <processReference refid="localJVM1" />
          <globusOption>
            <count>8</count>
            <maxTime>120</maxTime>
            <errorFile>
              ${GLOBUS_USER_HOME}/error.txt
            </errorFile>
          </globusOption>
        </globusProcess>
      </processDefinition>
    </processes>
```

```
  </infrastructure>
</ProActiveDescriptor>
```

- ARC (NorduGrid):

  ProActive provides org.objectweb.proactive.core.process.nordugrid.NGProcess to use such a protocol.

  To use ARC you will need to download the ARC Client[6].

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="/home/user/ProActive"/> <!--CHANGE ME!!!! -->
    <descriptorVariable name="JAVA_HOME"
      value="/path/to/jdk1.5.0" /><!-- Path of the remote JVM , CHANGE ME!!!! -->
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="plugtest" timeout="1200000"/>
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="plugtest">
        <jvmSet>
          <vmName value="Jvm1"/>

        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="ngProcess"/>
        </creation>
      </jvm>

    </jvms>
  </deployment>
  <fileTransferDefinitions>
 <fileTransfer id="ng_transfer">
  <file src="http://grid.uio.no/runtime/j2re1.4.2_08.tar.gz" dest="j2re1.4.2_08.tar.gz" />
  <file src="lib/ProActive.jar" dest="ProActive.jar" />
  <file src="lib/javassist.jar" dest="javassist.jar" />
  <file src="lib/components/fractal.jar" dest="fractal.jar" />
  <file src="lib/bouncycastle.jar" dest="bouncycastle.jar" />
  <file src="lib/log4j.jar" dest="log4j.jar" />
  <file src="lib/xercesImpl.jar" dest="xercesImpl.jar" />
```

[6] http://ftp.nordugrid.org/download/index.html

```
    <file src="dist/proactive-log4j" dest="proactive-log4j" />
    <file src="dist/proactive.java.policy" dest="proactive.java.policy" />
  </fileTransfer>
</fileTransferDefinitions>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM1">
        <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
          </classpath>
          <javaPath>
            <absolutePath value="${JAVA_HOME}/bin/java"/>
          </javaPath>
          <policyFile>
            <absolutePath value="proactive.java.policy"/>
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath value="proactive-log4j"/>
          </log4jpropertiesFile>
        </jvmProcess>
      </processDefinition>
      <processDefinition id="ngProcess">

<ngProcess class="org.objectweb.proactive.core.process.nordugrid.NGProcess" hostname="ng_frontend">
          <processReference refid="localJVM1"/>
          <fileTransferDeploy refid="ng_transfer">
            <copyProtocol>processDefault</copyProtocol>
            <sourceInfo prefix="file://${PROACTIVE_HOME}" />
          </fileTransferDeploy>
           <ngOption>
            <executable>
              <absolutePath value="${PROACTIVE_HOME}/scripts/unix/cluster/ngStartRuntime.sh"/>
            </executable>
            <count>28</count>
            <outputFile>hello.txt</outputFile>
            <errorFile>hello1.txt</errorFile>
           </ngOption>
        </ngProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

- MPI

  ProActive provides org.objectweb.proactive.core.process.mpi.MPIDependentProcess to use such a protocol. You have to couple this process with the DependentListProcessDecorator explained below.

Here is the complete example that you can find within the ProActive distribution.

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

  <variables>
    <descriptorVariable name="PROACTIVE_HOME" value="ProActive" />
    <descriptorVariable name="REMOTE_HOME" value="/home/smariani" />
    <descriptorVariable name="MPIRUN_PATH"
      value="/usr/src/redhat/BUILD/mpich-1.2.6/bin/mpirun" />
    <descriptorVariable name="QSUB_PATH"
      value="/opt/torque/bin/qsub" />
    <descriptorVariable name="USER_HOME"
      value="/user/smariani/home" />
  </variables>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="CPI" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="CPI">
        <jvmSet>
          <vmName value="Jvm1" />
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name="Jvm1">
        <creation>
          <processReference refid="sshProcess" />
        </creation>
      </jvm>
    </jvms>
  </deployment>
  <fileTransferDefinitions>
    <fileTransfer id="transfer">
      <!-- Transfer mpi program on remote host -->
      <file src="cpi" dest="cpi" />
    </fileTransfer>
  </fileTransferDefinitions>
  <infrastructure>
    <processes>

      <processDefinition id="localJVM1">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <classpath>
```

```
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
            <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
          </classpath>
          <javaPath>
            <absolutePath
              value="${REMOTE_HOME}/jdk1.5.0_05/bin/java" />
          </javaPath>
          <policyFile>
            <absolutePath
              value="${REMOTE_HOME}/proactive.java.policy" />
          </policyFile>
          <log4jpropertiesFile>
            <absolutePath
              value="${REMOTE_HOME}/${PROACTIVE_HOME}/dist/proactive-log4j" />
          </log4jpropertiesFile>
          <jvmParameters>
            <parameter
              value="-Dproactive.useIPaddress=true" />
            <parameter value="-Dproactive.rmi.port=6099" />
          </jvmParameters>
        </jvmProcess>
      </processDefinition>

      <!-- remote jvm Process -->
      <processDefinition id="jvmProcess">
        <jvmProcess
          class="org.objectweb.proactive.core.process.JVMNodeProcess">
          <jvmParameters>
            <parameter
              value="-Dproactive.useIPaddress=true" />
            <parameter value="-Dproactive.rmi.port=6099" />
          </jvmParameters>
        </jvmProcess>
      </processDefinition>

      <!-- pbs Process -->
      <processDefinition id="pbsProcess">
        <pbsProcess
          class="org.objectweb.proactive.core.process.pbs.PBSSubProcess">
          <processReference refid="localJVM1" />
          <commandPath value="${QSUB_PATH}" />
          <pbsOption>
            <hostsNumber>3</hostsNumber>
            <processorPerNode>1</processorPerNode>
            <bookingDuration>00:02:00</bookingDuration>
            <scriptPath>
              <absolutePath
                value="${REMOTE_HOME}/${PROACTIVE_HOME}/scripts/unix/cluster/
pbsStartRuntime.sh" />
```

```
            </scriptPath>
          </pbsOption>
        </pbsProcess>
      </processDefinition>

      <!-- mpi Process -->
      <processDefinition id="mpiCPI">
        <mpiProcess
          class="org.objectweb.proactive.core.process.mpi.MPIDependentProcess"
          mpiFileName="cpi">
          <commandPath value="${MPIRUN_PATH}" />
          <mpiOptions>
            <processNumber>3</processNumber>
            <localRelativePath>
              <relativePath origin="user.home"
                value="${PROACTIVE_HOME}/scripts/unix" />
            </localRelativePath>
            <remoteAbsolutePath>
              <absolutePath value="${REMOTE_HOME}/MyApp" />
            </remoteAbsolutePath>
          </mpiOptions>
        </mpiProcess>
      </processDefinition>

      <!-- dependent process -->
      <processDefinition id="dpsCPI">
        <dependentProcessSequence
          class="org.objectweb.proactive.core.process.DependentListProcess">
          <processReference refid="pbsProcess" />
          <processReference refid="mpiCPI" />
        </dependentProcessSequence>
      </processDefinition>

      <!-- ssh process -->
      <processDefinition id="sshProcess">
        <sshProcess
          class="org.objectweb.proactive.core.process.ssh.SSHProcess"
          hostname="nef.inria.fr" username="smariani">
          <processReference refid="dpsCPI" />
        </sshProcess>
      </processDefinition>
    </processes>
  </infrastructure>
</ProActiveDescriptor>
```

15.7.3. DependentListProcessDecorator

This process is used when a process is dependent on an another process. The first process of the list can be any process but the second one must be a DependentProcess. Thus the second one has to implement the org.objectweb.proactive.core.process.DependentProcess interface.

```
<!-- mpi Process -->
<processDefinition id="mpiCPI">
  <mpiProcess
```

```
      class="org.objectweb.proactive.core.process.mpi.MPIDependentProcess"
      mpiFileName="cpi">
      <commandPath value="${MPIRUN_PATH}" />
      <mpiOptions>
        <processNumber>3</processNumber>
        <localRelativePath>
          <relativePath origin="user.home"
            value="${PROACTIVE_HOME}/scripts/unix" />
        </localRelativePath>
        <remoteAbsolutePath>
          <absolutePath value="${REMOTE_HOME}/MyApp" />
        </remoteAbsolutePath>
      </mpiOptions>
    </mpiProcess>
</processDefinition>

<!-- dependent process -->
<processDefinition id="dpsCPI">
    <dependentProcessSequence
      class="org.objectweb.proactive.core.process.DependentListProcess">
      <processReference refid="pbsProcess" />
      <processReference refid="mpiCPI" />
    </dependentProcessSequence>
</processDefinition>

<!-- ssh process -->
<processDefinition id="sshProcess">
    <sshProcess
      class="org.objectweb.proactive.core.process.ssh.SSHProcess"
      hostname="nef.inria.fr" username="smariani">
      <processReference refid="dpsCPI" />
    </sshProcess>
</processDefinition>
```

We can notice in this example that the second process of the DependentListProcess (mpiCPI) instantiate the org.objectweb.proactive.core.process.mpi.MPIDependentProcess class which, as required above, implements the org.objectweb.proactive.core.process.DependentProcess interface.

15.8. Infrastructure and services

As mentionned previously, instead of creating jvms, ProActive gives the possibility to acquire existing jvms. To do so, as shown in the example below, a service must be referenced in the **acquisition** tag. At this point one service is implemented: **RMIRegistryLookup**. The RMIRegistryLookup service performs a lookup in an RMIRegistry at the **url specified in the service definition** to find a ProActiveRuntime (a JVM) with the given name.

```
<?xml version='1.0' encoding='UTF-8'?>
<ProActiveDescriptor
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='DescriptorSchema.xsd'>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name='VnTest' property='multiple'/>
    </virtualNodesDefinition>
```

```
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode='VnTest'>
        <jvmSet>
          <vmName value='Jvm1'/>
        </jvmSet>
      </map>
    </mapping>
    <jvms>
      <jvm name='Jvm1'>
        <acquisition>
          <serviceReference refid='lookupRMI'/>
        </acquisition>
      </jvm>
    </jvms>
  </deployment>
  <infrastructure>
    <services>
      <serviceDefinition id='lookupRMI'>
        <RMIRegistryLookup url='//localhost:2020/PA_JVM1'/>
      </serviceDefinition>
    </services>
  </infrastructure>
</ProActiveDescriptor>
```

The **RMIRegistryLookup** service needs only an **URL** to perform the lookup.

The example above shows a VirtualNode **VnTest**, that is mapped to one JVM, **Jvm1**. **Jvm1** represents a JVM that will be acquired using an RMI Lookup.

Fault Tolerance can also be defined at the service level. See Chapter 32. Fault-Tolerance[7] for more information.

15.9. Processes

ProActive provides also the ability to use all processes defined above without using XML Deployment Descriptor. You can programmatically create such processes.

In order to get familiar on how to create processes programmatically, see the javadoc of the org.objectweb.proactive.core.process package[8].

For instance, you can create an SSH process as follows:

```
SSHProcess ssh = new SSHProcess(new SimpleExternalProcess("ls -lsa"));
ssh.setHostname("kisscool.inria.fr");
ssh.startProcess();
```

This piece of code will create an SSHProcess in charge of executing the ls -lsa command on kisscool.inria.fr.

[7] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../AdvancedFeatures/multiple_html/FaultTolerance.html

[8] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/ReferenceManual/pdf/../../api_complete/org/objectweb/proactive/core/process/package-summary.html

15.10. Descriptor File Transfer

As it is explained in Chapter 16, *ProActive File Transfer*, ProActive provides a File Transfer mechanism which enables to transfer a file from a node to another one. File transfers are normally done through the ProActive API. however, File Transfers can also be specified using ProActive Descriptors. The main advantage of this scheme is that it allows deployment and retrieval of input and output (files). In this section we will concentrate on mainly three topics:

- XML Descriptor File Transfer Tags
- Deployment File Transfer
- Retrieval File Transfer

15.10.1. XML Descriptor File Transfer Tags

The File Transfer related tags, are placed inside the descriptor at three different parts (or levels).

The first one corresponds to the **fileTransferDefinitions** tag, which contains a list of FileTransfer definitions. A FileTransfer definition is a high level representation of the File Transfer, containing mainly the file names. It is created in such a way, that no low level information such as: hosts, protocols, prefix is present (this is the role of the low level representation). The following example shows a FileTranfer definition named **example** which has to placed between the *deployment* and the *infrastructure* tag:

```
<fileTransferDefinitions>
  <fileTransfer id="mytransfer">
    <file src="src-test.txt" dest="dest-test-deployment.txt" />
  </fileTransfer>
  <!--
  <fileTransfer id="another_transfer">
    ...
  </fileTransfer>
   -->
</fileTransferDefinitions>
```

The FileTransfer definitions can be referenced through their names, from the **VirtualNode** tags using two attributes:**fileTransferDeploy** and **fileTransferRetrieve**. The first one, corresponds to the file transfer that will take place at deployment time, and the second one corresponds to the file transfer that the user will trigger once the user application is done.

```
<virtualNode name="VN" fileTransferDeploy="mytransfer" fileTransferRetrieve="mytransfer"/>
```

All the low level information such as: hosts, username, protocols, prefix, etc... is declared inside each process. Both **fileTransferDeploy** and **fileTransferRetrieve** are specified separately using a **refid** attribute. The **refid** can be a direct reference to a FileTransfer definition, or the keyword **implicit**. If **implicit** is used, then the reference will be inherited from the corresponding VirtualNode. In the following example both mechanisms (Deploy and Retrieve) reference indirectly and directly the example definition:

```
<processDefinition id="ssh_jily">
  <sshProcess
    class="org.objectweb.proactive.core.process.ssh.SSHProcess"
    hostname="jily.inria.fr">
    <processReference refid="localJVM" />

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

```
    <fileTransferDeploy refid="implicit">
      <copyProtocol>processDefault, rcp, scp, pft</copyProtocol>
      <sourceInfo prefix="/tmp"/>
      <destinationInfo prefix="/tmp"/>
    </fileTransferDeploy>

    <fileTransferRetrieve refid="implicit">
      <sourceInfo prefix="/tmp"/>
      <destinationInfo prefix="/tmp"/>
    </fileTransferRetrieve>

  </sshProcess>
</processDefinition>
```

In the example above, **fileTransferDeploy** has an implicit refid. This means that the File Transfer definitions used will be inherited from the VirtualNode. The first element shown inside this tag corresponds to **copyProtocol**. The **copyProtocol** tag specified the sequence of protocols that will be executed to achieve the FileTransfer at deployment time. Notice the **processDefault** keyword, which specifies the usage of the default copy protocol associated with this process. In the case of the example, this corresponds to an **sshProcess** and therefore the Secure Copy Protocol (scp) will be tried first. To complement the higher level File Transfer definition, other information can be specified as attributes in the **sourceInfo** and **destinationInfo** elements. In this example, we provide a **prefix** attribute that indicates from and to which directory the file should be transferred. Other attributes such as **hostname** and **username** can also be given.

For **fileTransferRetrieve**, no copyProtocol needs to be specified. ProActive will use its internal mechanism to transfer the files. This implies that no **hostname** or **username** are required.

15.10.2. Supported protocols for file transfer deployment

The supported protocols for file transfer are the following one:

- **pftp** (ProActive File Transfer Protocol)
- **scp** (ssh processDefault)
- **rcp** (rsh processDefault)
- **nordugrid** (Nordugrid processDefault)

15.10.3. Triggering File Transfer Deploy

The start of the File Transfer will take place when the deployment of the descriptor file is executed. In the case of **external protocols** (**scp, rcp**), this will take place before the process deployment. In the case of **internal protocols** (**nordugrid**), this will take place with the process deployment. In any case, it should be noted that interesting things can be achieved, such as transfering the ProActive libraries into the deploying machine using an **on-the-fly** style. This means that it is possible to deploy on remote machines without having ProActive **pre-installed**. Even further, when the network allows, it is also possible to transfer other required libraries like the JRE (Java Runtime Environment).

There is one protocol that behaves differently from the others: the ProActive FileTransfer Protocol (**pftp**). **pftp** uses the ProActive FileTranfer API (described in Chapter 16, *ProActive File Transfer*), to transfer files between nodes. The main advantage of using the **pftp** is that no external copy protocols are required to transfer files at deployment time. Therefore, if the grid infrastructure does not provide a way to transfer files, a FileTransfer Deploy can still take place using the **pftp**. On the other hand, the main drawback of using **pftp** is that ProActive must already be install on the remote machines, and thus **on-the-fly** deployment is not possible.

15.10.4. Triggering File Transfer Retrieve

Since distributed application's termination is difficult to detect. The responsibility of triggering the deployment corresponds to the user. To achieve this, we have provided a specific method that will trigger the retrieval of all files associated with a VirtualNode.

```
List<RemoteFile> rfList = virtualNode.getVirtualNodeInternal().fileTransferRetrieve();
```

This will trigger the retrieval of all the files specified in the descriptor, from all the nodes that were deployed using this virtual node using the **pftp**.

As a result of calling this method, a list of RemoteFile will be created, representing all the retrieved files.

15.10.5. Advanced: FileTransfer Design

This section provides internal details and information on how the File Transfer is implemented. Reading this section to use the File Transfer mechanisms provided by ProActive is not necessary.

15.10.5.1. Abstract Definition (High level)

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

- **id** attribute - A unique identification name.
- **file** element - source and optionally destination file name.
- **dir** element - source and optionally destination directory name.

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

15.10.5.2. Concrete Definition (Low level)

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

- **refid** attribute: A reference to an abstract definition, or the "**implicit**" key word indicating that the reference will be inherited from the VirtualNode.
- **copyProtocols** element: A sequence of Copy Protocols that will be used.
- **sourceInfo** and **destinationInfo** element: 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, information contained in the protocol can be overridden if specified.

15.10.5.3. How Deployment File Transfer Works

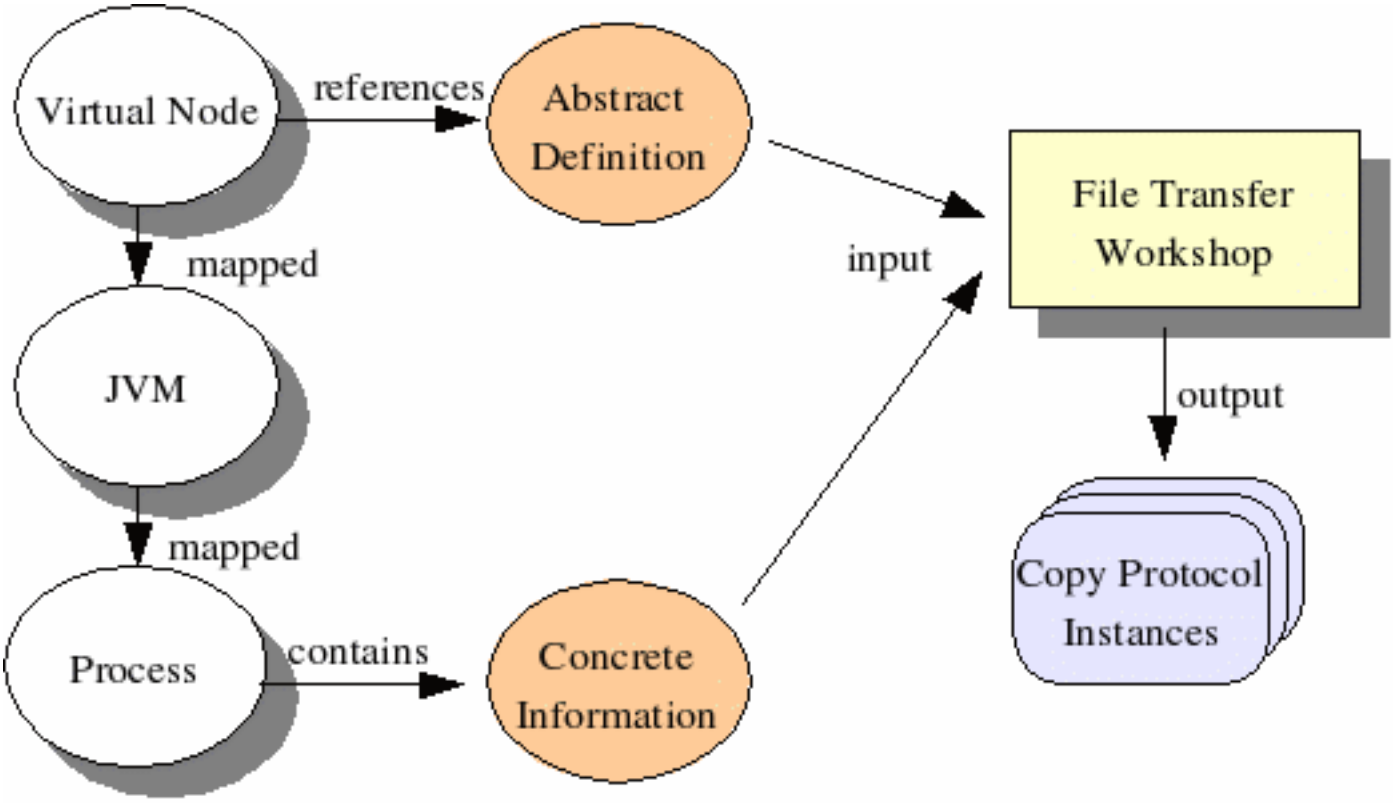


Figure 15.1. File Transfer Design

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

Each CopyProtocol will be tried before the deployment takes place, until one succeeds. Once one succeeded or all failed, the process deployment will take place.

Chapter 16. ProActive File Transfer

16.1. Introduction and Concepts

ProActive provides a way to transfer files or directories from on node to another. There are two kind transfers:

- **Push** - Transfer from the local node to a remote node.
- **Pull** - Transfer from a remote node to the local node.
- **Third part transfer** - Transfer from a remote node to another remote node.

The transfer can take place at any of the following moments:

- **Deployment** Time: At the beginning of the application to input the data.
- **Retrieval** Time: At the end of the application to collect results.
- **During the user application**: To transfer information between nodes.

Warning

If you use the GCM Deployment, the transfer can only be made by the user at the third moment. Contrary to the XML Deployment which provides tags for transferring file at deployment and retrieval times, GCM Deployment is not able to do that. If you want to know how to transfer files at deployment and retrieval times, please refer to Section 15.10, “Descriptor File Transfer”.

16.2. File Transfer API

16.2.1. API Definition

ProActive File Transfer is done through the org.objectweb.proactive.api.PAFileTransfer class which implements the following methods:

```
public static RemoteFile pull(Node srcNode, File srcFile, File dstFile) throws IOException;
public static List<RemoteFile> pull(Node srcNode, File[] srcFile, File[] dstFile) throws IOException;

public static RemoteFile push(File srcFile, Node dstNode, File dstFile) throws IOException;
public static List<RemoteFile> push(File[] srcFile, Node dstNode, File[] dstFile) throws IOException;

public static RemoteFile transfer(Node srcNode, File srcFile, Node dstNode, File dstFile);
public static List<RemoteFile> transfer(Node srcNode, File[] srcFile, Node dstNode, File[] dstFile);
public static List<RemoteFile> transfer(Node srcNode, File[] srcFile, Node dstNode, File[] dstFile, int bsize, int
 numFlyingBlocks) throws IOException;

public static RemoteFile mkdirs(Node node, File path) throws IOException;
```

These methods handle the transfer of file between ProActive **Nodes**. The **pull** methods retrieve a file/directory located on a remote machine to the local machine. The **push** methods transfer a file/directory available on the local node to the specified remote node. The **transfer** methods allow to transfer between third part nodes. The mkdirs method creates a directory on the remote machine.

The file transfers are performed in an asynchronous fashion. Each of these methods returns a **RemoteFile** object which represents the file transfer operation and the remote file's location. When these methods are invoked a **RemoteFile** instance is immediately returned, before the file transfer operation has been completed. The **RemoteFile** object provides a way to monitor the status of the file transfer:

```
/**
 * This class represents the result of a file transfer operation.
 *
 * When a file transfer operation is invoked, for example by using the
```

```
 * {@link org.objectweb.proactive.api.PAFileTransfer  PAFileTransfer} API, a RemoteFile instance
 * is returned. A RemoteFile can be used, among others, to determine
 * if the file transfer operation has been completed, to wait for the
 * file transfer operation to finish, to obtain a reference on the node
 * where the file is stored, etc...
 *
 * Additionally, new file transfer operations can be triggered from
 * this class, for example to push/pull a remote file to another node.
 */
@PublicAPI
public interface RemoteFile extends Serializable {

    /**
     * @return true if the file transfer operation that spawned this RemoteFile instance has finished, or an exception. false otherwise.
     */
    public boolean isFinished();

    /**
     * This method blocks the calling thread until the file transfer operation is
     * finished, or failed. If the operation failed, the exception is raised.
     * @throws IOException The cause of the file transfer operation failure.
     */
    public void waitFor() throws IOException;

    /**
     * Pulls the remote file represented by this instance into the local destination.
     * The result of this operation yields a new RemoteFile instance.
     *
     * @param localDst The local destination where the file will be stored.
     * @return A new RemoteFile instance representing the file transfer operation on the local node.
     * @throws IOException  If an error is detected during the initialization phase of the file transfer.
     */
    public RemoteFile pull(File localDst) throws IOException;

    /**
     * Push the RemoteFile represented by this instance to the Node and File location provided as parameters.
     * @param dstNode The destination node.
     * @param dstRemote The destination file on the destination node.
     * @return A new RemoteFile instance representing this file transfer operation.
     * @throws IOException If an error was detected during the initialization phase of the file transfer.
     */
    public RemoteFile push(Node dstNode, File dstRemote) throws IOException;

    /**
     * @return The node where the file is stored (or was meant to be stored, if an error took place).
     */
    public Node getRemoteNode();

    /**
     * @return The destination File where the data is stored on the remote node (or was meant to be stored, if an error took place).
     */
    public File getRemoteFilePath();
```

```
    /**
     * Deletes a remote file or directory recursively (i.e. deletes non-empty directories) represented by this RemoteFile
 instance.
     * This methods is synchronous.
     *
     * @return true if the delete was successful, false otherwise.
     * @throws IOException If an error was encountered while deleting the file
     */
    public boolean delete() throws IOException;

    /**
     * Queries the existence of a RemoteFile. This method is synchronous (blocking).
     *
     * @return true if the remote file exists, false otherwise.
     * @throws IOException If an error is encountered while querying the remote file.
     */
    public boolean exists() throws IOException;

    /**
     * Queries if the RemoteFile is a directory. This method is synchronous (blocking).
     *
     * @return true if the remote file is a directory, false otherwise.
     * @throws IOException If an error is encountered while querying the remote file.
     */
    public boolean isDirectory() throws IOException;

    /**
     * Queries if the RemoteFile is a File. This method is synchronous (blocking).
     *
     * @return true if the remote file is a file, false otherwise
     * @throws IOException If an error is encountered while querying the remote file.
     */
    public boolean isFile() throws IOException;
}
```

The **isFinished** and **waitFor** can be invoked to query and wait on the file transfer status. The **pull** method can be used to fetch the **RemoteFile** from the remote node, into the local **Node**, and the **push** methods can be used to send the **RemoteFile** to another **Node**.

16.2.2. How to use the API Example

The following class shows how to use the ProActive File transfer whether it be with the GCM or the XML deployments. Actually, once the nodes concerned by the transfer have been retrieved, there is no difference between these two deployment. The only difference is how to get a node.

```
/**
 * @author ffonteno
 *
 * This class has been made to test the ProActive File Transfer and
 * extract code snippets for the documentation.
 *
 */
public class FileTransferTest {
```

```
/**
 * Returns the virtual node whose name is VNName and described in the GCM descriptor file whose
 * path is descriptorPath
 *
 * @param descriptorPath path of the GCM descriptor file
 * @param VNName name of the virtual node
 * @return the virtual node whose name is VNName and described in the GCM descriptor file whose
 * path is descriptorPath
 */
public static GCMVirtualNode getGCMVirtualNode(String descriptorPath, String VNName) {
    // Retrieves the file corresponding to your application descriptor
    File applicationDescriptor = new File(descriptorPath);

    GCMApplication gcmad;
    try {

        // Loads the application descriptor file
        gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);

        // Starts the deployment
        gcmad.startDeployment();

        GCMVirtualNode vn = gcmad.getVirtualNode(VNName);
        vn.waitReady();

        return vn;

    } catch (ProActiveException e) {
        e.printStackTrace();
        return null;
    }
}

/**
 * Copies the file whose path is sourcePath on the local host to the file with path destPath
 * on each machine that hosts a node mapped with the virtual node VNName
 *
 * @param descriptorPath path of the GCM descriptor file
 * @param VNName name of the virtual node
 * @param sourcePath source path
 * @param destPath destination path
 * @throws IOException
 * @throws NodeException
 */
public static void testGCMFileTransfer(String descriptorPath, String VNName, String sourcePath,
        String destPath) throws IOException, NodeException {

    Node srcNode = NodeFactory.getDefaultNode();
    System.out.println(srcNode.getVMInformation().getHostName());

    GCMVirtualNode vn = FileTransferTest.getGCMVirtualNode(descriptorPath, VNName);
    long nbNodes = vn.getNbCurrentNodes();

    for (long l = 0; l < nbNodes; l++) {
```

```
            System.out.println("Node number " + l);
            Node destNode = vn.getANode();
            System.out.println(destNode.getVMInformation().getHostName());
            RemoteFile rf = PAFileTransfer.transfer(srcNode, new File(sourcePath), destNode, new File(
                destPath));
            rf.waitFor();
            System.out.println(rf.getRemoteFilePath().getPath());
        }
    }

    /**
     * Returns the virtual node whose name is VNName and described in the XML descriptor file whose
     * path is descriptorPath
     *
     * @param descriptorPath path of the XML descriptor file
     * @param VNName name of the virtual node
     * @return the virtual node whose name is VNName and described in the GCM descriptor file whose
     * path is descriptorPath
     * @throws ProActiveException
     */
    public static VirtualNode getXMLVirtualNode(String descriptorPath, String VNName)
            throws ProActiveException {

        // Creates the ProActiveDescriptor corresponding to the descriptor file
        ProActiveDescriptor proActiveDescriptor = PADeployment.getProactiveDescriptor(descriptorPath);

        // Gets the virtual node named VN1 described in the descriptor file.
        VirtualNode virtualNode = proActiveDescriptor.getVirtualNode(VNName);

        // Activates the virtual node.
        // For activating several virtual node at once, you can use
        // proActiveDescriptorAgent.activateMappings()
        virtualNode.activate();

        return virtualNode;
    }

    /**
     * Copies the file whose path is sourcePath on the local host to the file with path destPath
     * on each machine that hosts a node mapped with the virtual node VNName
     *
     * @param descriptorPath path of the XML descriptor file
     * @param VNName name of the virtual node
     * @param sourcePath source path
     * @param destPath destination path
     * @throws IOException
     * @throws ProActiveException
     */
    public static void testXMLFileTransfer(String descriptorPath, String VNName, String sourcePath,
            String destPath) throws IOException, ProActiveException {

        Node srcNode = NodeFactory.getDefaultNode();
        System.out.println(srcNode.getVMInformation().getHostName());
```

```
        VirtualNode virtualNode = FileTransferTest.getXMLVirtualNode(descriptorPath, VNName);

        long nbNodes = virtualNode.getNbMappedNodes();

        for (long l = 0; l < nbNodes; l++) {
            System.out.println("Node number " + l);
            Node destNode = virtualNode.getNode();
            System.out.println(destNode.getVMInformation().getHostName());
            RemoteFile rf = PAFileTransfer.transfer(srcNode, new File(sourcePath), destNode, new File(
                destPath));
            rf.waitFor();
            System.out.println(rf.getRemoteFilePath().getPath());
        }

    }

    /**
     * Test ProActive File Transfer with the two deployments.
     *
     * @param args should be: (GCMA.xml|Descriptor.xml) VirtualNodeName sourcePath destPath
     * @throws ProActiveException
     */
    public static void main(String[] args) throws ProActiveException {
        try {
            if (args.length < 4) {
                System.out.println("Wrong number of arguments");
                System.out
                        .println("Usage: java FileTransferTest (GCMA.xml|Descriptor.xml) VirtualNodeName sourcePath
destPath");
            }
            //testGCMFileTransfer(args[0], args[1], args[2], args[3]);
            testXMLFileTransfer(args[0], args[1], args[2], args[3]);
        } catch (NodeException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

This example uses a third part transfer but, given this example, using a push or pull transfer is really straight forward.

How to obtain a local Node

A local node be easyly obtained by using the NodeFactory:

```
Node srcNode = NodeFactory.getDefaultNode();
```

How to obtain a Node from an Active Object reference

The Node where an Active Object resides can be obtained as follows:

```
Object o  =  PAActiveObject.newActive(...);
...
Node node = PAActiveObject.getActiveObjectNode(o);
```

16.2.3. How File Transfer API works

The File Transfer API is built on top of ProActive active object and future file asynchronism model. When pulling or pushing a file from a Node, two service Active Objects (AO) are created. One is placed on the local machine and the other one on the remote site. The file is then split into blocks, and transferred over the network using remote invocations between these two AOs.

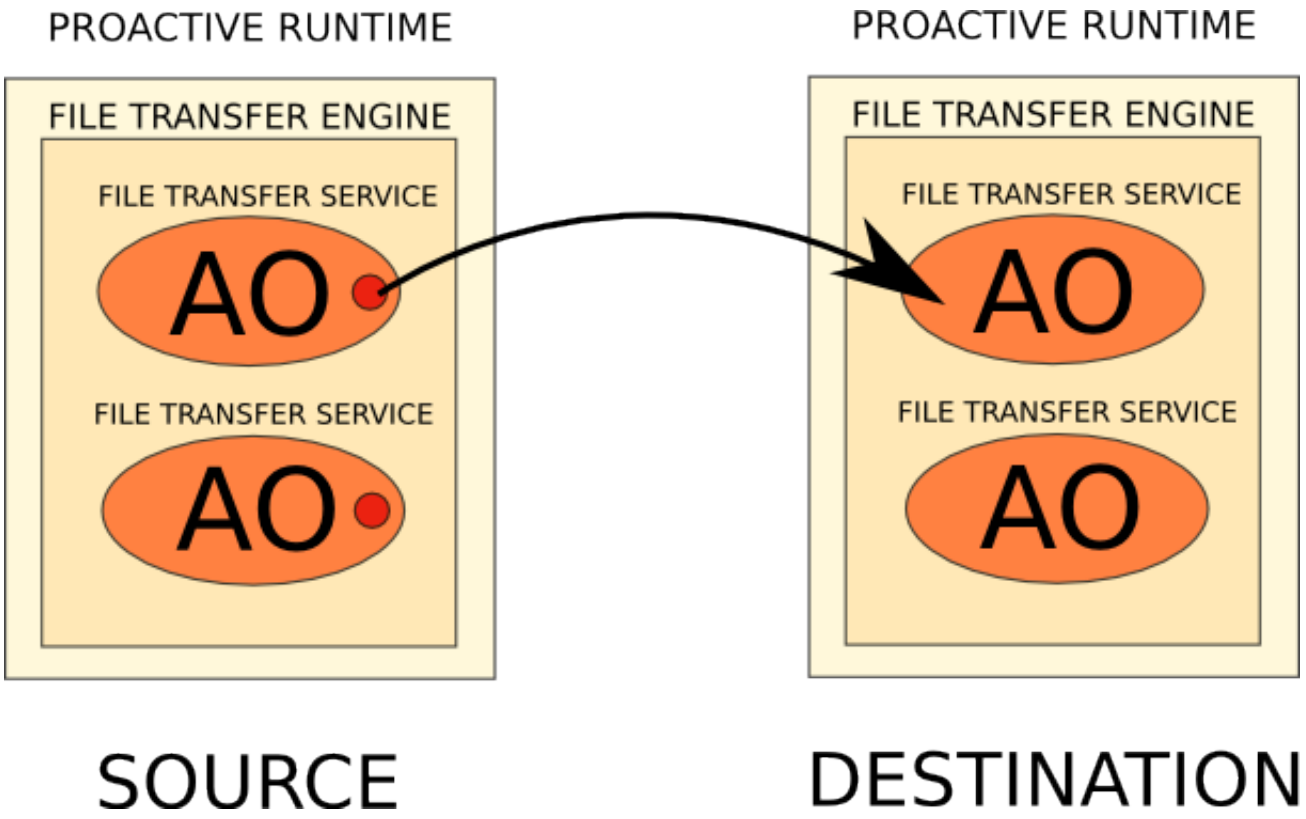


Figure 16.1. File Transfer Representation

Each ProActive runtime has a unique FileTransferEngine which manages several FileTransferService. These services are active object and are in charge of transferring files. When a transfer is launch, the unique FileTransferEngine asks one of its FileTransferService to send the file to another FileTransferService which is a FileTransferService of the destination node. The source FileTransferService then splits the file into blocks and calls the saveFileBlock method on the destination FileTransferService to write each block in the destination node.

Chapter 17. How to terminate a ProActive application

17.1. Destroying active objects

In the master slave paradigm, the master should wait for all of its slaves to be known. Then all active objects can be destroyed by letting the master terminate all the slaves and then itself.

In a more complex topology, a distributed termination algorithm must be designed to coordinate the *PAActiveObject.terminateActiveObject()* method calls.

The simplest alternative is to enable the **Distributed Garbage Collector (DGC)** by setting the **proactive.dgc** property to **true** and let it destroy unreferenced active objects and cycles of idle active objects.
The DGC is the bridge between active objects and the local GC. It automatically calls the PAActiveObject.terminateActiveObject() method on unreachable active objects so that you do not have to.

The default configuration should be suitable in most case, but it is possible to alter the timing of the DGC algorithm using two parameters: **proactive.dgc.ttb** and **proactive.dgc.tta**.

proactive.dgc.ttb specifies the period of the heartbeat in milliseconds, and **proactive.dgc.tta** specifies the time to wait in milliseconds before assuming a beat was received from all referencers.

Scientifically speaking, **proactive.dgc.tta** must be much greater than **proactive.dgc.ttb** to account for the network latency. By default, **proactive.dgc.ttb** is 30 seconds and **proactive.dgc.tta** is 150 seconds.

When using the DGC, it's possible that because of references kept in user threads, garbage is not found as quickly as desired.
To this end, the **PALifeCycle.userThreadTerminated()** method should be called at the end of the main thread. This instructs the DGC that at this point, active objects referenced in user threads will not be used anymore.

Both explicit termination (calling terminateActiveObject()) and implicit termination (letting the DGC call it) trigger the call of the potentially endActivity() method in the terminated active object.

17.2. Killing JVMs

Once active objects are destroyed, JVMs must be killed in order to complete the cleanup. Alternatively, one can focus on killing the JVMs, that will get rid of the active objects too.

Depending on which deployment you have used (GCM or XML Deployment), this action can be done by calling a method on the descriptor object.

17.2.1. Killing JVMs started with a GCM Deployment

If you have used the GCM Deployment to deploy your active objects on several hosts, you can kill the deployed JVMs as well as theirs hosted active objects by calling the *kill()* method on the descriptor object.

```
GCMApplication gcmad = PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
.....................
gcmad.kill();
```

This terminates all the ProActive Runtimes that have been started by this Application.

17.2.2. Killing JVMs started with an XML Deployment

If you have used the XML Deployment to deploy your active objects on several hosts, you can kill the deployed JVMs as well as theirs hosted active objects by calling the *killall(boolean softly)* method on the descriptor object.

```
//---------- Returns a ProActiveDescriptor object from the xml file
ProActiveDescriptor pad = PADeployment.getProactiveDescriptor(String xmlFileLocation);
pad.activateMappings();

...

//---------- Kills every jvms deployed with the descriptor
pad.killall(false);
```

If **softly** is set to false, then all JVMs created when activating the descriptor are killed abruptly. If true, then the JVM that has originated the creation of a rmi registry waits until registry is empty before dying. To be more precise a thread is created to ask periodically the registry if objects are still registered.

Chapter 18. Variable Contracts for Descriptors

18.1. Variable Contracts for Descriptors

18.1.1. Principle

The objective of this feature is to allow the use of variables into GCM descriptors and 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 useful inside a descriptor because they can factorize frequent parameters (for example, a variable like ${PROACTIVE_HOME} can be defined, set and used in a descriptor). But also, because they can be used to establish a contract between the program and the descriptor.

18.1.2. Variable Types

Variables can be set in more than one place. The following table describes where each variable type can be defined. When the value is set on multiple places, then the definition specified in the priority column will take precedence. In the priority column, items towards the left have more priority.

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	-	JavaProperty
javaPropertyDescriptorDefault	JavaProperty, Descriptor, Program	Program	JavaProperty, Descriptor, Program
javaPropertyProgramDefault	JavaProperty, Descriptor, Program	Descriptor	JavaProperty, Program, Descriptor

Table 18.1. Variable Types

18.1.3. Variable Types User Guide

In order to help to 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 whereas the deployer corresponds to the responsible of writing the deployment descriptor. The variables represent rights and responsibilities between the two parties (contract) as specified in the following table:

Type	Behavior	When to use this type
descriptorVariable	The value has to be set in the descriptor, and cannot be specified in the program.	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 has to be set in the program, and cannot be specified in the descriptor.	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 has to be specified in the descriptor. The programmer has the ability to change the value in the program. If the value is changed in the program, then this new value will have precedence over the one defined in the descriptor.	The programmer may override the default value, but the responsibility of setting a default belongs to the deployer.
programDefaultVariable	A default value has to be specified in the program. The descriptor has the ability to change the value. If the value is changed in the descriptor, then this new value will have precedence over the one defined in the program.	The deployer may override the default value, but the responsibility of setting a default belongs to the programmer.
javaPropertyVariable	Takes the value from the corresponding Java property.	When a variable will only be known at runtime through the Java properties, and no default has to be provided by the descriptor or the application.
javaPropertyDescriptorDefault	Takes the value from the corresponding java property. A default value can also be set from the descriptor or the program. If no property is found, the descriptor default value will override the program default value.	When the descriptor sets a default value, that can be overridden at deployment using a java property.
javaPropertyProgramDefault	Takes the value from the corresponding java property. A default value can also be set from the program or the descriptor. If no property is found, the program default value will override the program default value	When the program sets a default value, than can be overridden at deployment using a java property.

Table 18.2. Variable behaviors and use cases

18.1.4. Variables Example

18.1.4.1. Descriptor Variables

The following example shows how to define and use a variable into a GCM descriptor. All variables has to be set in an environment section.

```
<environment>
  <descriptorVariable name="PROACTIVE_LIB" value="dist/lib"/>
  <javaPropertyVariable name="proactive.home" />
  <descriptorDefaultVariable name="NUMBER_OF_VIRTUAL_NODES" value="20"/>
  <programVariable name="VIRTUAL_NODE_NAME"/>
  <javaPropertyVariable name="java.home"/>
  <javaPropertyDescriptorDefault name="host.name" value="localhost"/>
  <javaPropertyProgramDefault name="priority.queue"/>
  <javaPropertyVariable name="user.home" />
  <descriptorVariable name="hostCapacity" value="2"/>
  <descriptorVariable name="vmCapacity" value="2"/>
```

```
  <!-- Include external variables from files-->
  <includePropertyFile location="file.properties"/>
</environment>


<application>
  <proactive base="root" relpath="${proactive.home}">
    <configuration>
      <applicationClasspath>
        <!-- Use example -->
        <pathElement base="root" relpath="${proactive.home}/${PROACTIVE_LIB}/ProActive_examples.jar"/>
      </applicationClasspath>
    </configuration>
    <!-- ... -->
</proactive>
</application>
```

As for the XML descriptor version, all variables has to 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="20"/>
  <programVariable name="VIRTUAL_NODE_NAME"/>
  <javaPropertyVariable name="java.home"/>
  <javaPropertyDescriptorDefault name="host.name" value="localhost"/>
  <javaPropertyProgramDefault name="priority.queue"/>

  <!-- 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>
...
```

18.1.4.2. Program Variables

Variables can easily be defined into the Java program in the following way:

```
VariableContractImpl variableContract = new VariableContractImpl();
variableContract.setVariableFromProgram("VIRTUAL_NODE_NAME", "testnode",
      VariableContractType.ProgramVariable);
variableContract.setVariableFromProgram("NUMBER_OF_VIRTUAL_NODES", "10",
      VariableContractType.DescriptorDefaultVariable);
variableContract.setVariableFromProgram("priority.queue", "vip",
      VariableContractType.JavaPropertyProgramDefault);

File descriptor = new File(gcmaFile);
GCMApplication gcma;
try {
```

```
    gcma = PAGCMDeployment.loadApplicationDescriptor(descriptor, variableContract);

    //Usage example
    VariableContract vc = gcma.getVariableContract();
    String proActiveHome = vc.getValue("PROACTIVE_HOME");

} catch (ProActiveException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

In you are using the previous deployment, you should proceed as follows:

```
XML_LOCATION="/home/user/descriptor.xml";
VariableContract variableContract= new VariableContractImpl();
variableContract.setVariableFromProgram( "VIRTUAL_NODE_NAME", "testnode",
 VariableContractType.ProgramVariable);
variableContract.setVariableFromProgram( "NUMBER_OF_VIRTUAL_NODES", "10",
 VariableContractType.DescriptorDefaultVariable);
variableContract.setVariableFromProgram( "priority.queue", "vip",
 VariableContractType.JavaPropertyProgramDefault);
ProActiveDescriptor pad = PADeployment.getProactiveDescriptor(XML_LOCATION, variableContract);

//Usage example
VariableContract vc=pad.getVariableContract;
String proActiveHome=vc.getValue("PROACTIVE_HOME");
```

18.1.5. External Variable Definitions Files

Instead of declaring variables into your program or into your descriptor, it is also possible to define them into an external file and to make reference to it in your description. This can be very useful if you want the same variables for several descriptors.

18.1.5.1. Properties Files

This approach uses Sun microsystems properties file format[1]. The format is plain text with one definition per line in the format **variable** = **value**, as shown in the following example:

```
# 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 (=).

18.1.5.2. XML Files

In the previous deployment version, it was also possible to define variables into an XML file. It has been removed for the GCM deployment thinking it was useless.

```
<!-- 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"/>
```

[1] http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html#load(java.io.InputStream)

```
</variables>
```

18.1.6. Program Variable API

18.1.6.1. Relevant import packages

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

18.1.6.2. Available Variable Types

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

18.1.6.3. 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 19. GCMDeployment and Virtual Environment.

Hardware Virtualization has reached an important point of interest in the datacenter and the workstation. Whether you want to save money by saving power, footprint or cooling system, or improve reactivity, fault tolerance and availability of your services, you will probably use hardware virtualization capabilities. You can also improve desktop management and delivery by using hardware virtualization for your workstations. That's because we can see the hardware virtualization as a fast growing and widespread technology that ProActive must leverage such infrastructures.

19.1. ProActive & Hardware Virtualization QuickStart.

This section gives a brief introduction about how hardware virtualization works with the different kind of virtualization software we can meet and how to provide seamless ProActive deployment on a virtual enabled infrastructure.

19.1.1. Hardware Virtualization Overview.

Hardware virtualization allows to run several operating systems on a unique machine. This is done thanks to a specific software called “Virtual Machine Monitor” (vmm). Every virtualization solutions needs a particular operating system to work (“Dom0” in case of bare metal vmm and “host OS” in case hosted virtualization, we will dig deeper in this later). The vmm can either emulates specific hardware devices or grants access to real hardware to the virtual machine (or guest operating system ie. vm). One thus benefits many features for different purposes. We identify 2 kinds of vmm:

- **Type 1 hypervisor** (or bare metal).

  This type of hypervisor is named "bare metal" because it doesn't need any operating system beneath to work. Depending on what kind of product you are using, you will benefit different drivers for different hardware and, maybe, won't be able to get the software functional because of unsupported hardware (see a description at: http://en.wikipedia.org/wiki/Hypervisor). The most often, you'll need a “DOM 0” virtual machine, which is in fact the operating system that stands for your workspace, to manage your virtual infrastructure. The virtual machine monitor (VMM) itself is a small footprint software (about 50Mo) which will only check and schedule underlying hardware access. To set up your virtual environment you need an extra software (xm for XenOss, xe for XenServer, vmx for ESX ...) which is usable directly from your DOM 0. This type of hypervisor is, in general, more efficient and faster than others hypervisors as it implements its own hardware access policy at the lowest possible level. Thus, you completely avoid overhead induced by an underlying operating system. Here are some well-known hypervisors: XenServer[1], Hyper-V[2], xVM Server[3], VMware ESX/ESXi[4], Xen OSS[5]

- **Type 2 hypervisor** (or hosted).

  This type of hypervisor is running on top of an operating system and is seen as a common process so does any virtual machine container. We can thus measure the host operating system overhead that treat the guest operating system as a child process whereas it was more a "brother" in the case of type one hypervisor. Furthermore, every kind of virtualization process cannot be used in hosted virtualization for some reasons (browse http://en.wikipedia.org/wiki/Platform_virtualization for more details). Here are the main type two hypervisors: Virtualbox[6], VMware Server[7], KVM[8]

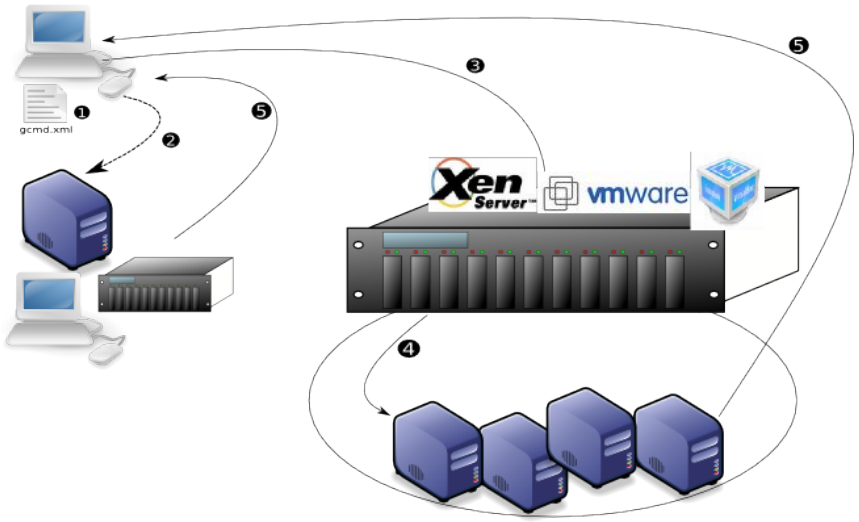
The most important hardware emulation a ProActive user has to be informed of is the network part. We can essentially distinguish three sorts of network provisions:

[1] http://community.citrix.com/cdn/xs
[2] http://www.microsoft.com/hyper-v-server/en/us/default.aspx
[3] http://www.sun.com/software/products/xvmserver/index.xml
[4] http://www.vmware.com/products/vi/esx/
[5] http://www.xen.org/
[6] http://www.virtualbox.org/
[7] http://www.vmware.com/products/server/
[8] http://www.qumranet.com/

- **Bridge Networking.** Here, the network routing is made at the 3rd level of the OSI/ISO stack. Your computer's NIC bound to your company/internet network is setup in “PROMISCUOUS MODE” to intercept not only packet intended for the host/dom0 IP but also for newly created Virtual interfaces designed to provide network to virtual machines. Depending on what kind of virtualization product you use, you may have to create the virtual interfaces by yourself (brctl/openvpn on linux systems & Network Manager on Windows). With this solution, your guest operating systems are part of the company network like your host computer. That means a fully point-to-point connectivity between both hardware/virtual machine on your network.
- **Nat/Route.** The network routing is also made at the 3rd level of the OSI/ISO stack. This time, a newly created virtual interface will ensure POST/PRE-ROUTING & MASQUERADE for your virtual machine to have network access. It is really easy to find such configuration example on the internet (Here for linux for instance[9]). The introduction of the Network address translation (NAT) adds a difficulty for ProActive deployment and makes the virtual machines belonging to the subnet hidden for the outer network. Nevertheless you can deploy from your host within the virtual machines binding ProActive on the NATing interface.
- **NAT user.** This time network routing is made at the virtualization software layer. For the outer world (the host machine comprised) the virtual machines are unreachable. The most often, the associated subnet has the pattern 10.0.0.1/24, 10.0.0.1 & 10.0.0.2 are the IPs for the host machine (only seen by the virtual machines) and every virtual machines belongs to its own subnet (virtual machines can't speak to each other). Notice that it is IMPOSSIBLE, without ProActive Message Routing, to deploy on a such network configuration.

19.1.2. How does it work with ProActive.

For conventional ProActive deployment, the “deployer” builds the needed commands to contact machines specified in the xml GCMDeployment Descriptor and launches a ProActive Runtime. This one next registers to its parent to be able to create Nodes for your application. Because when using the virtualization layer one adds an intermediate (the VMM) and maybe binds virtual machines to NATed network, the boot process can't be the same. This diagram shows the differences:



ProActive deployment on virtual infrastructure.

- **1**: The “deployer” reads the deployment descriptor and builds the commands to connect to the specified machines.
- **2**: It uses the built commands to connect and starts child ProActive Runtime. This part is only executed if you deploy on hardware infrastructures.
- **3**: The virtual machine managers are identified in the GCMDeployment descriptor, contacted by the deployer to start the chosen virtual machines. The vmm identifies/build the virtual environment to deploy and start the corresponding virtual machines. Every virtual machine needs some information to be able to bootstrap ProActive environment, to be able to provide such information, a dedicated servlet handles virtual machines' requests. When hitting the good html page, the virtual machine gets the compulsory pieces of information. To get that done, the vmm update the virtual machine environment with the servlet's url (this part of the job is done thanks to the underlying virtualization software layer).

[9] http://www.revsys.com/writings/quicktips/nat.html

- **4**: At boot time, the virtual machine launch a daemon that tries to identify the virtual environment within the one it is running. Once that is done, one gets the servlet's url, gets the parent ProActive Runtime's url, the deployment ID and other useful information, and boots the child ProActive Runtime.
- **5**: The child runtimes are booted. They register to their parent and create Nodes for the ProActive Application.

19.1.3. Software compatibility.

The table below lists the different supported virtualization software, the virtualization type and the associated section to use in GCMDeployment descriptor to use if you want to deploy such infrastructures. To get more information on a particular VMM you can refer to the wanted subsection.

Vendor	Product	Virtualization type	GCMD tag
VMware	Server<2, Workstation<6.5	Hosted	vmware-vix
	Server > 2	Hosted	vmware-vi
	ESX/ESXi > 2.5	Bare Metal	vmware-vi
Citrix	XenServer	Bare Metal	libxen
Xen.org	Xen OSE	Bare Metal	libvirt
Sun	Virtualbox	hosted	virtualbox-ws
Open Source Community	Qemu - KVM - LXC - OpenVZ	Hosted	libvirt
Microsoft	Hyper-V	Bare Metal	hyper-v winrm
Microsoft	Hyper-V	Bare Metal	hyper-v wmi

19.2. Virtualization Layer Setup.

To be able to deploy ProActive Application (and more generally virtualized infrastructure from GCMDeployment descriptor), you have to setup your virtual infrastructure by hand and test the setup a first time. Currently, we do not give the capability to dynamically register or build virtual machines, you only can boot, clone (for some provider) and destroy your envrionment. This section describes the overall/common prerequisites (compulsory steps you have to go through not depending on a given virtualization software).

19.2.1. Overall prerequisites.

You must setup the virtualization software. This begins by choosing the kind of vmm you want to use depending on your needs. You have essentially have to focus on the kind of operating system you want to virtualize, the kind of hardware you have, the type of network to provide for your virtual machines to access network, and if performances are a critical requirement for your application.

The more often and at the time this documentation was written (07/02/2009), the bare metal hypervisors don't allow advanced settings for network. The main solution is to deploy a dedicated virtual machine that will handle network traffic for NAT-ing or Bride-ing (that can represents a major flaw). Bare metal VMMs don't support many hardware, you have to ensure your configuration is compatible (depending on what kind of guest os you want to run, you also may need specific hardware. For instance, using XenServer for Windows virtualization, it is compulsory to own an intel vt, vtx or vtx2 chip. Besides, notice that for compliant kernel in case of paravirtualization[10] , you have to use modified guest operating system). And finally, the main products are designed for datacenter virtualization or advanced desktop delivery, they may lack some important features like hierarchical snapshots, removable devices handling, sounds... The main advantage to run type one hypervisors is that they expose better performances and benefit fast pace evolution due to hard pressure from industry client users. Hosted virtualization software are more desktop user friendly and often used for desktop/client virtualization, critical software development... They always expose at least NAT/Bridge networking capabilities and do not require to run modified guest operating system. The main flaw is that they run pretty much slower than type one hypervisor due to host operating system overhead...

Once you have chosen the kind of VMM you want to use, you have to “deploy” (here, “deploy” has a special meaning. For virtual infrastructure managers it means “register your virtual machines”). Just create a virtual machine referring to your virtualization software documentation. Once this is done, if you want to deploy ProActive applications, you need to register the provided python scripts as daemons for your guest operating system. To register a new daemon on your guest, you have to focus on the operating system. For

[10] http://en.wikipedia.org/wiki/Paravirtualization

instance, with windows NT core based systems you can use autoexnt.exe/autoexnt.bat[11] that registers as a service. This application runs the SYSROOT/windows32/autoexnt.bat content at system launch, for example:

```
set python_path=C:\\Python26\python.exe
set pa_home=C:\\ProActive
set log_file=C:\\log\proactive-runtime.log
"%python_path%" "%pa_home%\scripts\virtualization\proactivemain.py" "%log_file%"
```

For a Debian based system, you can use update-rc.d: (cp all virtualization scripts in /etc/init.d)

```
sudo update-rc.d proactive-runtime start 99 2 3 5 . stop 99 0 1 6 .
```

Here is the content of /etc/init.d/proactive-runtime

```
#!/bin/sh
#
# This is a basic dummy shell script to get your proactive runtime bootstraping using
# the proactive python API.
# It can be used on debian like distros. Have it registered using update-rc.d...
#
# Basic support for IRIX style chkconfig
# chkconfig: 235 99 99
# description: Manages the services needed to run ProActive

# Basic support for the Linux Standard Base Specification 1.0.0 (to be used by
# insserv for example)
### BEGIN INIT INFO
# Provides: ProActive
# Required-Start: VBoxControl vmware-guestd
# Required-Stop:
# Default-Start: 2 3 5
# Default-Stop: 0 1 6
# Description: Manages the services needed to run ProActive
### END INIT INFO

# BEGINNING_OF_UTIL_DOT_SH
#!/bin/sh
#
#
# Get lsb functions
. /lib/lsb/init-functions
. /etc/default/rcS

PIDFILE=/var/run/proactive-runtime.pid
LOGFILE=/var/log/proactive-runtime.log
#change the path of you python and proactive apps here
EXE="/usr/bin/python /etc/init.d/proactivemain.py ${LOGFILE}"
ARGS=
```

[11] http://support.microsoft.com/kb/243486/fr

```
case "$1" in
  start)
    [ -e ${LOGFILE} ] || touch ${LOGFILE}
    log_begin_msg "Starting ProActive-Runtime..."
    start-stop-daemon --start --exec ${EXE} -b -m --pidfile ${PIDFILE} -- ${ARGS}
    log_end_msg $?
    ;;
  stop)
    log_begin_msg "Stopping ProActive-Runtime..."
    start-stop-daemon --stop -p ${PIDFILE}
    log_end_msg $?
    ;;
  restart)
    $0 stop
    sleep 1
    $0 start
    ;;
  *)
    log_success_msg "Usage: /etc/init.d/proactive-runtime {start|stop|restart}"
    exit 1
esac

exit 0
```

For redhat based systems, you'll have to use chkconfig utility and replace start-stop-daemon by a background launch plus remove log messages.

```
#!/bin/sh
#
# This is a basic dummy shell script to get your proactive runtime bootstraping using
# the proactive python API.
# It can be used on debian like distros. Have it registered using update-rc.d...
#
# Basic support for IRIX style chkconfig
# chkconfig: 235 99 99
# description: Manages the services needed to run ProActive

# Basic support for the Linux Standard Base Specification 1.0.0 (to be used by
# insserv for example)
### BEGIN INIT INFO
# Provides: ProActive
# Required-Start: VBoxControl vmware-guestd
# Required-Stop:
# Default-Start: 2 3 5
# Default-Stop: 0 1 6
# Description: Manages the services needed to run ProActive
### END INIT INFO

PIDFILE=/var/run/proactive-runtime.pid
LOGFILE=/var/log/proactive-runtime.log
#change the path of you python and proactive apps here
```

```
EXE="/usr/bin/python /etc/init.d/proactivemain.py ${LOGFILE}"
ARGS=

case "$1" in
  start)
    [ -e ${LOGFILE} ] || touch ${LOGFILE}
    echo "Starting ProActive-Runtime..."
    ${EXE}& ${ARGS}
    if [ $? -eq 0 ]; then echo success; else echo fail; fi
    ;;
  stop)
    echo "Stopping ProActive-Runtime..."
    pkill proactive
    if [ $? -eq 0 ]; then echo success; else echo fail; fi
    ;;
  restart)
    $0 stop
    sleep 1
    $0 start
    ;;
  *)
    echo "Usage: /etc/init.d/proactive-runtime {start|stop|restart}"
    exit 1
esac

exit 0
```

As you can see in that short snippet, the python script that holds the main entry point of the daemon is proactivemain.py. This application accepts zero or one parameter. If you supply one parameter, it will be the file that the daemon will log into. If you don't provide anything as parameter, the program will log on stdout. This script tries to find in which virtualized environment it is running and thus load the associated python functions module. These modules are used to find/retrieve the bootstrap servlet url that is able to serve the needed pieces of information to bootstrap ProActive Runtime. Those python scripts and daemon examples are located in %ProActive_Home %/scripts/virtualization.

19.2.2. Editor dependent.

Here are listed the steps to follow in case of GCM deployment and for a given virtualization software.

19.2.2.1. VMware

These software products allow ProActive deployment from GCM.

For all handled VMware products you have to install VMware guest tools. This step is really straight forward. For VMware Server prior 2, power the virtual machine on, on the VMware Server console, click Devices, install guest tools. This mounts a CD into your VM. If you run a Windows OS, launch the installer.exe, and on a Linux one, run the installer.sh. For VMware Server > 2, launch your virtual machine, from the virtual machine administration page, right column, click install guest tools. This will also mount a CD within your VM. For VMware ESX/ESXi, follow the same step from the Vi Client software and idem for Workstation client. The tools installed are used to allow the ProActive Runtime daemon to read the bootstrap servlet's http page that will serve the information needed to boot.

19.2.2.1.1. VMware ESX/ESXi

ESX is currently the killer software at VMware and is declined in different version depending on what kind of support you subscribed. However you can use the last ESXi which is pretty akin to ESX except that it is free and comes with less features and hardware support. Both ESX and ESXi can be purchased in “installable and embedded” versions. The embedded release is distributed by OEM vendors

with their servers solution hence you don't have to bother with it. You can see it as a firmware settled on the hardware flash memory. It only contains the drivers it needs to work and no more. You can't imagine make it run on an other platform than the one it is made for. The installable release comes with more hardware support and can be installed on a common hard disk or directly boot from a USB key. See http://www.vm-help.com/ and user guide for more information about that. Finally, once you have your ESX installed and working, you don't need anything else to be able to use GCM to deploy.

Once the hypervisor is installed, you have several choices to give your virtual machines access the network. With legacy ESX releases, one virtual machine (really small) was dedicated to routing network and thus provided a NAT network. Note that that technique is used with several other type one hypervisors. With latest ESX releases you can use virtual switch and other features to minimize the impact on your company network.

19.2.2.1.2. VMware Server and Workstation

VMware Server (formerly GSX) is a major free software for server virtualization as type two hypervisor at VMware's. Depending on what release of VMware Server you are running you'll have to focus on different things.

- **VMware Server < 2.** The VMware Server releases prior to 2 are shipped with a rich client (VMware Server Console) to be able to connect to every compatible server.This means that you need two different software solution to manage your virtual environment. If you want to use that product: First, create a new user belonging to the vmware group, this user will be the administrator for your system. Download the software from VMware web site[12] and install it. The install is straight forward, the only thing you have to focus on is setup. Prior to launch your VMware Server for the first time you have to run a configuration script, “vmware-config”, that will fix several useful information. Among those one, the previously created user for administration, directory for virtual machine disks storage, directory for documentation, libraries etc... and the port for remote authentication. Set this port and remember its value, we will need it at the deployment time.
- **VMware Server >= 2.** VMware Server after version 2 are VMware Virtual Infrastructure[13] compliant. They come with a web user interface for management (allow end user to remotely manage his virtual environment). If you want to use this software first create a new user belonging to the vmware group. This user will be the administrator of your virtual environment. Then download VMware Server from http://www.vmware.com/products/server/ and install it. Prior to launch VMware Server for the first time you have to run a perl script, vmware-config, to setup your VMware environment. During that step, the most important thing to do is to remember the authentication port you set. We may need it at the deployment time.

19.2.2.2. Hyper-V

This software allows ProActive deployment.

Microsoft released its bare metal hypervisor, Microsoft Hyper-V, with its server operation system Windows Server 2008 and 2008 R2. Microsoft is a real competitor in the Virtualization ecosystem for the well known XenServer and VMware ESX. It currently supports 2 different ways to be managed using Microsoft's DCOM protocol[14] or http with the Microsoft's implementation of WSMAN[15] named WINRM. If you decide to use DCOM, you must set the firewall rules and if you want to use http, you must setup the WINRM environment. For the latter, you can refer to this tutorial[16]

19.2.2.3. XenServer

This software allows ProActive deployment.

XenServer is the property of Citrix[17] and is based on the free and open source Xen OSS[18]. It is a type one hypervisor compatible with several OEM and which is said to be faster than VMware's equivalent products as it uses paravirtualization technique[19] (whereas VMware uses full-virtualization - binary translation[20]). To use it, just refer to the user guide[21] for installation. You can also install it

[12] http://www.vmware.com/support/server/doc/releasenotes_server.html
[13] http://www.vmware.com/products/vi/
[14] http://en.wikipedia.org/wiki/Distributed_Component_Object_Model
[15] http://en.wikipedia.org/wiki/WS-Management
[16] http://blogs.dirteam.com/blogs/sanderberkouwer/archive/2008/02/23/remotely-managing-your-server-core-using-winrm-and-winrs.aspx
[17] http://www.citrix.fr/
[18] http://www.xen.org/
[19] http://en.wikipedia.org/wiki/Paravirtualization
[20] http://en.wikipedia.org/wiki/Full_virtualization
[21] http://www.citrix.com/lang/English/lp/lp_1688615.asp

on an external usb device thanks to that tutorial[22]. If you want enable efficient cloning feature for your XenServer environment, be sure to use an ext3 file system and nothing else. If you use an LVM base repository, when cloning a virtual machine you won't benefit the Copy on Write (COW) feature that allows two virtual machines to share a common backing disk file and saves changes in separated files. An ext3 base repository does. To change the backing file system of your virtual machine pool try this tutorial[23]

The way we provide ProActive deployment for XenServer virtualization platform is different than for other virtualization software. The implementation of the virtualization layer embeds a small database to handle infrastructure information. We use that database to store and retrieve the bootstrap servlet's url to be able to get needed information to boot ProActive Runtime. To store the information, we tag a VM's datastore space with its NIC MAC address (this information is used as key, that's why we need to ensure every virtual machine doesn't own a similar MAC address. If you don't contrast XenServer settings, it ensures that every assigned NIC MAC addresses are different.). When the ProActive Runtime Daemon starts from the virtual machine, it first tries to retrieve all declared MAC addresses, connects to XenServer datastore and iterates all virtual machines' space to retrieve the tagged MAC address and finally get the needed pieces of information. To be able to get that done, you need to precise XenServer's url, user and password inside the xenserver.py file.

19.2.2.4. Virtualbox

This software allows ProActive deployment.

Virtualbox is a type two hypervisor created by Innotek and since February 2008 property of Sun Microsystems. This virtualization software is currently the only one running on Linux, Windows, MacOS and Solaris and is a precursor of some virtualization techniques such as seamless virtualization and 3D hardware virtualization. At the moment, only Virtualbox non open source edition is shipped with vboxwebsrv application which is compulsory to use virtualization features at deployment time. Be sure to install a version of Virtualbox that comes with that add-on. The install process is really straight forward and the software easy to use, see the project web site for more information[24].

For ProActive deployment, we highly recommend to use Virtualbox with NAT network where all the network trafic is emulated by the VMM at the software level (no impact on the company network when installing/deploying) and ProActive Message Routing Protocol (PAMR). You can use other network settings but we don't ensure the deployment will work. You also have to start vboxwebsrv. You can use personal settings, the only one with which you have to be careful is the session time out. We encourage a session time out of 20s. We also notices that some vboxwebsrv releases had some troubles with authentication. If it is your case, just submit the following command:

```
VBoxManage setproperty websrvauthlibrary null
```

and then restart the vboxwebsrv program.

19.2.2.5. Xen Open Source

This hypervisor is not supported for ProActive Deployment. You need libvirt to make Xen Oss work at the deployment time.

Xen OSS is the base component of XenServer. It is a type one hypervisor which comes as a particular kernel for several linux distribution (redhat, fedora, ubuntu ...). The best way to have a fully personalized Xen is to compile sources and build its own kernel. If you want to use that software: Download the source tarball[25], CAREFULY read the readme (you also can use that tutorial[26]) and build your own kernel(s):

```
cd /usr/src
  wget http://bits.xensource.com/oss-xen/release/3.3.1/xen-3.3.1.tar.gz
  tar xvf xen-3.3.1.tar.gz
  ln -s xen-3.3.1 xen
```

[22] http://community.citrix.com/blogs/citrite/dannyw/2009/03/12/Installing+XenServer+on+an+USB+Drive
[23] http://www.tokeshi.com/index.php?option=com_content&task=view&id=5025
[24] http://www.virtualbox.org/
[25] http://www.xen.org/download/
[26] http://www.howtoforge.org/debian_etch_xen_3.1

```
cd xen
```

Now we compile Xen with 2 targets kernels

```
make world vmxassist=y KERNELS="linux-2.6-xen0 linux-2.6-xenU"
install.sh
```

The vmxassist=y flag means that we want to allow HVM (Hardware-enhanced Virtual Machine, requires hardware suppport) guests to run, so we ask Xen to build a specific container that will be available in /usr/lib/xen/boot/hvmloader. To be able to compile such a Kernel, you must ensure that the packages glibc-devel, dev86, libvncserver,SDL and SDL-devel are installed. If you want, you can make them more efficient...

```
cd /usr/src/xen/build-linux-2.6.18.8-xen0_[arch]
make menuconfig
```

Check the following options:

```
Xen --> [*] Privileged Guest (domain 0)
File systems --> [*] Quota support
        [M] Old quota format support
        [M] Quota format v2 support

Device Drivers ---> Networking support --> [M] Dummy net driver support

Networking support --> Networking options --> [*] Network packet filtering (replaces ipchains) --> IP: Netfilter
Configuration --> [M] IP tables support (required for filtering/masq/NAT).
```

then:

```
make
make modules
make modules_install
make install
```

Just repeat the whole procedure unchecking

```
Xen --> [ ] Privileged Guest (domain 0)
```

for the domU kernel. Note that you can forget that last build as the last releases of Xen OSS allow to run both host and guest with the same kernel config and the same efficiency. Just be sure that the modified /boot/grub/menu.lst is up to date and reboot on your newly created dom0 install

Once your Xen Oss virtual environment is installed you have to setup your virtual machines inside. To help you, you have several client software, among those one: Convirt (aka Xenman) and virt-manager. Our favorite one for the installation step is Convirt[27]. It

[27] http://www.convirture.com/

will allow you to install both paravirtualized and hardware assisted virtualized guest operating systems (the setup "by hand is feasible" for paravirtualized but more complicated for others and deals with Xen configuration files, thus we encourage to avoid it...). Here is one of the good tutorial[28] you can find on the web. Virt-manager is better for virtual machine management and handles more virtual environments (furthermore it is built on top of libvirt API[29]).

19.2.2.6. KVM, Qemu-KVM, Qemu

This hypervisors are not supported for ProActive Deployment. You need to use libvirt in addition to make these software products work with deployment.

That hypervisors are all based on Qemu[30]. They are all type two hypervisors that allow emulation[31] (Qemu) and hardware assisted virtualization[32](Qemu-KVM and KVM). The best advantage with using that softwares (in fact, only Qemu is concerned) is that it is possible to use them entirely in user space (don't need root account for installation and use). If you want to use that solution, download one of the products (Qemu-KVM is an add-on for Qemu): Qemu[33] or KVM[34] and install it thanks to the README explanations.

To install virtual machines and manage your environment you can use Convirt (aka Xenman)[35] or virt-manager[36] or try to get it done by hand.

```
dd if=/dev/zero of=disk.img bs=512k count=1 seek=2000
mkfs.ext2 disk.img
```

build a raw disk image with a capacity of 1Go and an ext2 file system (for linux based systems).

```
kvm -net nic -net user -m 512 -boot d -cdrom /home/jmguilla/isos/fedora.iso disk.img
```

install your OS thanks to the .iso file

```
kvm -net nic -net user -m 512 -boot c disk.img
```

to boot your virtual machine once the operating system is installed.

19.2.2.7. Libvirt

Libvirt (see http://libvirt.org/) is a management API project for virtualized infrastructures. This aim is to provide an unique API / Interface to manage several VMM. You can notice that it currently supports: Xen (on Linux and Solaris), QEMU, KVM, LXC, OpenVZ, User Mode Linux, VirtualBox, and that other supports are announced (VMware server and ESX).

If you used Virt-manager to install and set up your virtualized environment, you must already have Libvirt installed on your computer and your virtual machines registered. Otherwise, download the project here[37] and install it. You then have to register your previously created virtual machines. You can read the associated documentation[38] to get it done.

[28] http://blog.fedora-fr.org/smootherfrogz/post/XenMan
[29] http://libvirt.org/
[30] http://www.nongnu.org/qemu/
[31] http://en.wikipedia.org/wiki/Emulator
[32] http://en.wikipedia.org/wiki/Hardware-assisted_virtualization
[33] http://www.nongnu.org/qemu/download.html
[34] http://sourceforge.net/project/showfiles.php?group_id=180599&package_id=209008
[35] http://www.convirture.com/
[36] http://virt-manager.et.redhat.com/download.html
[37] http://libvirt.org/downloads.html
[38] http://libvirt.org/docs.html

When evaluating this environment, we noticed different "hot points" about which one you have to be careful. Libvirt may use different mates[39] for authentication, just be sure that you have enable the good user with sufficient permissions and using the good authentication mechanism. Be sure to understand the basic knowledge about URLs[40]. Understand that, when using Xen Oss, you'll have to perform extra configuration on the server side to set up your Xen VMM the good way (to use either sockets, http, legacy remote control ...).

19.3. GCMDeployment and Virtual Environment.

This section describes how to use virtualization deployment capabilities from the GCM deployment descriptor file.

19.3.1. Principles.

The GCM deployment descriptor allows to deploy virtualized infrastructures. If you want to leverage this feature, you first have to declare the virtual appliances you want to deploy, by what kind of VMM they are managed, and every other pieces of information required to get the job done (depending on the VMM you use, you may supply different amount of data...). This part is made in the “<infrastructure>” section. Here is an example:

```
<vms>
  <vmware-vi id="vmware-vi">
    <hypervisor url="https://jily.activeeon.com:8333/sdk"/>
    <hypervisor url="http://excalibur.activeeon.com:8222/sdk"/>
    <hypervisor url="http://bud.activeeon.com:18965/sdk"/>
    <authentication user="inria" pwd="inria123"/>
    <image key="myUbuntu" os="unix" count="2"/>
    <image key="myDebian" os="unix" />
    <image key="myGentoo" os="unix" />
  </vmware-vi>
  <libxen id="xenserver">
    <hypervisor url="http://192.168.1.166"/>
    <authentication user="root" pwd="root123"/>
    <image key="myUbuntu" os="unix" count="2"/>
  </libxen>
</vms>
```

The “<vms>” tag means that one defines a virtual manager section, every authorized child elements are software dependent and requires different information. Please refer to the section corresponding to your virtualization software to have more information. Nevertheless, these child elements have common sections:

- **hypervisor**

  This tag is optional, if you omit it, we'll try to deploy locally (note that in the case the uri contains a specific protocol, you'll have to specify the url even if connecting locally). This tag accept the following attributes: url. The format of the url is software dependent.

  This tag hasn't got any child element.
- **authentication**

  This tag accept the following attributes: user, pwd. A user with sufficient permissions to manage the virtual environment and his password.

  This tag hasn't got any child element.
- **image**

[39] http://libvirt.org/auth.html
[40] http://libvirt.org/uri.html

This tag accept the following attributes: key, os, count. **key** is the id of the virtual machine within your environment, the more often its name. **os** is the os type of the guest operating system, either windows or unix. **count** is not available for all software and specify the number of virtual machine of that type you want to boot. See the section matching your environment for more information.

This tag hasn't got any child element.

Once you defined your virtual infrastructure, you have to specify that you want to effectively “deploy” it by referencing the associated hypervisor section from the “<resources>” tag. For instance, with the code snippet above it will be:

```
<resources>
   <hypervisor refid="vmware-vi"/>
   <hypervisor refid="xenserver"/>
</resources>
```

In this section, you only declare the virtual resources you effectively want to deploy.

19.3.2. VMware products.

This section is suited for every VMware products.

19.3.2.1. Server<2, Workstation<6.5

This section doesn't supports clone. You need to reference every Virtualizing/src/vmware-vix/lib/ jars in the classpath. This section uses JNA[41] to bring the API to JAVA framework. Even if this section supports VMware server 2, we encourage you to use vmware-vi instead.

The more often, you can choose to install vmware-vix during installation of your VMware product. If you don't, download the last release HERE[42] and install it (needs at least 1.6 to work). Besides, you need to put the JNA's jar in your classpath and to fix the jna.library.path or java.library.path system property (launch the JVM with -Djna.library.path=...) to point to the libvixAllProducts.so lib, or specify it thanks to LD_LIBRARY_PATH variable on linux or PATH on windows.

This implementation is based on VMware Vix[43].

To deploy your virtual infrastructure with compliant products, you can use a <vmware-vix> section (see the associated section in the previous chapter). Here is an example on how to use it:

```
<vms>
   <vmware-vix id="vmware-vix" service="server" port="902">
      <hypervisor url="slave1.activeeon.com"/>
      <authentication user="inria" pwd="inria123"/>
      <image key="/home/jmguilla/VMware/Machines/myUbuntu/myUbuntu.vmx" os="unix"/>
   </vmware-vix>
<vms>
```

The <vmware-vix> element is a <vms> child element. It allows the following attributes:

- **id**

  A unique id for the entire GCM descriptor file. Used to be referenced from the resources section.

[41] http://jna.java.net/
[42] http://www.vmware.com/support/developer/vix-api/
[43] http://www.vmware.com/support/developer/vix-api/vix16_reference/

- **service**

  One of: server, vi, workstation to match your environment.
- **port**

  The authentication daemon port you set during installation.

The <vmware-vix> element allows the following child elements:

- **hypervisor**

  Here, the url attribute is the raw hostname or IP address of the host running the virtualization software (without any protocol or port).
- **authentication**

  VMware user with sufficient permissions.
- **image**

  The key attribute is the vmx file path describing the virtual machine you want to deploy.

  Count attribute not supported.

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is software dependent. If you provide a hypervisor without any url or no hypervisor, one tries to connect locally (but we encourage you to precise the service's url even if connecting locally, at least to specify protocol and port). You can put how many hypervisors you want.

19.3.2.2. Server>=2, ESX/ESXi>=2.5 and VI compliant products.

This section concerns every VMware products Virtual Infrastructure >= 2.0 compliant. You need to reference every Virtualizing/src/vmware-vi/lib/ jars in the classpath.

To deploy your virtual infrastructure with compliant products, you can use a <vmware-vi> section. Here is an example on how to use it:

```
<vms>
<vmware-vi id="vmware-vi">
   <hypervisor url="https://jily.activeeon.com:8333/sdk"/>
   <hypervisor url="http://excalibur.activeeon.com:8222/sdk"/>
   <hypervisor url="http://bud.activeeon.com:18965/sdk"/>
   <authentication user="inria" pwd="inria123"/>
   <image key="myUbuntu" os="unix" count="2"/>
   <image key="myDebian" os="unix" />
   <image key="myGentoo" os="unix" />
</vmware-vi>
<vms>
```

The <vmware-vi> tag says that one want to use the VMware Virtual Infrastructure module, it has the following child elements:

- **hypervisor**

  Here, the url attribute is the raw hostname or IP address of the host running the virtualization software (without any protocol or port).
- **authentication**

  VMware user with sufficient permissions.

- **image**

  The key attribute is the vmx file path describing the virtual machine you want to deploy.

  Count attribute not supported.

and the following attribute:

```
id="someID"
```

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is software dependent. If you provide a hypervisor without any url or no hypervisor, one tries to connect locally (but we encourage you to precise the service's url even if connecting locally, at least to specify protocol and port). You can put how many hypervisors you want.

The <authentication> tag is optional if connecting locally and is used to specify a user with sufficient permissions to manage your virtual environment through the VI interface. If you connect locally, you may avoid this tag. The only attributes of that section are user and pwd.

The <image> tag points out which virtual machines you want to deploy. It admits the following attributes:

```
key="someName"
os="windows"
count="3"
```

The “key” attribute designates the unique name of your virtual machine. Not every software force to fix a unique name per virtual machine (because you almost always manage your virtual infrastructure thanks to an user interface and can “click” the good one.), we ask you to do so to ensure the good function of the deployment feature. “os” attribute indicates the guest virtual machine's operating system. “count” allows you to precise the number of virtual machine of this type you want to deploy. To be able to ensure the feature for every VMware Virtual Infrastructure compliant software, we in fact emulate the clone feature. One just creates a perfect clone (regarding non persistent resources) of the template virtual machine and shares the persistent resources with it in an non-persistent way. That means that the clones will be able to use the parent virtual hard disk, but won't change it, once the virtual clones are powered off, every changes made to the disk will be lost. Because the hard disk is shared between several machines, you have to ensure that the exclusive lock can be taken. Indeed, the template virtual machine (which has exclusive write access on the disk) must be powered off to allows clones to be powered on.

19.3.3. Hyper-V

This section concerns Hyper-V using either WINRM or WMI. You need to reference every Virtualizing/src/hyperv-(winrm|wmi)/lib/ jars in the classpath.

To deploy your virtual infrastructure with Microsoft Hyper-V, you can use a <hyperv-winrm> or <hyperv-wmi> section (see the associated section in the previous chapter). Here is an example on how to use it:

```
<vms>
  <hyperv-winrm id="hyperv">
    <hypervisor url="http://192.168.1.166"/>
    <authentication user="root" pwd="root123"/>
    <image key="myUbuntu" os="unix" count="2"/>
  </hyperv-winrm>
<vms>
```

The <hyperv-winrm> (or hyperv-wmi) tag says that one want to use the module built on winrm API (or wmi) (see associated section in previous chapter), it has the following child elements:

```
<hypervisor>
<authentication>
<image>
```

and the following attribute:

```
id="someID"
```

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is service dependent. If you use wmi, just provide the FQDN (or IP), if you use winrm, provide a URL matching your winrm configuration (https://myserver.com for instance.). Note that in the latter case, if you decide to use https, you must setup the "javax.net.ssl*" properties. If you provide a hypervisor without any url or no hypervisor, one tries to connect locally (but we encourage you to precise the service's url even if connecting locally, at least to specify protocol and port). You can put how many hypervisors you want.

The <authentication> tag is optional if connecting locally and is used to specify a user with sufficient permissions to manage your virtual environment. If you connect locally, you may avoid this tag. The only attributes of that section are user and pwd.

The <image> tag points out which virtual machines you want to deploy. It admits the following attributes:

```
key="someName"
os="windows"
count="3"
```

The “key” attribute designates the unique name of your virtual machine. Not every software force to fix a unique name per virtual machine (because you almost always manage your virtual infrastructure thanks to an user interface and can “click” the good one.), we ask you to do so to ensure the good function of the deployment feature. “os” attribute indicates the guest virtual machine's operating system. “count” allows you to precise the number of virtual machine of this type you want to deploy. The clone feature for Microsoft Hyper-V uses Copy-On-Write, that means that the clone virtual machines use differencing hard disks to store modification of the cloned disk (work like snapshots, you cannot use the cloned disk anymore without violating the clone disk integrity.)

19.3.4. XenServer

This section concerns products compatible with LibXen (SDK for web service access for XenServer). Note that the associated python script launcher doesn't work with python 3 due to XenAPI.py incompatibility. You need to reference every Virtualizing/src/xenserver/lib/ jars in the classpath.

To deploy your virtual infrastructure with compliant products, you can use a <libxen> section (see the associated section in the previous chapter). Here is an example on how to use it:

```
<vms>
  <libxen id="xenserver">
    <hypervisor url="http://192.168.1.166"/>
    <authentication user="root" pwd="root123"/>
    <image key="myUbuntu" os="unix" count="2"/>
```

```
    </libxen>
<vms>
```

The <libxen> tag says that one want to use the module built on LibXen API (see associated section in previous chapter), it has the following child elements:

```
<hypervisor>
<authentication>
<image>
```

and the following attribute:

```
id="someID"
```

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is service dependent. If you provide a hypervisor without any url or no hypervisor, one tries to connect locally (but we encourage you to precise the service's url even if connecting locally, at least to specify protocol and port). You can put how many hypervisors you want.

The <authentication> tag is optional if connecting locally and is used to specify a user with sufficient permissions to manage your virtual environment through XenCenter, xe or xm. If you connect locally, you may avoid this tag. The only attributes of that section are user and pwd.

The <image> tag points out which virtual machines you want to deploy. It admits the following attributes:

```
key="someName"
os="windows"
count="3"
```

The “key” attribute designates the unique name of your virtual machine. Not every software force to fix a unique name per virtual machine (because you almost always manage your virtual infrastructure thanks to an user interface and can “click” the good one.), we ask you to do so to ensure the good function of the deployment feature. “os” attribute indicates the guest virtual machine's operating system. “count” allows you to precise the number of virtual machine of this type you want to deploy. To enable an efficient clone feature on XenServer, be sure to host your virtual machines on an EXT based pool. If you want to change the backing file system of your virtual machine disk pool, refer to that tutorial[44].

19.3.5. KVM, Qemu, Qemu-KVM, LXC, UML, Xen OSS

This section concerns every Libvirt compliant software. This module doesn't allow ProActive deployment. You need to reference every Virtualizing/src/libvirt/lib/ jars in the classpath. This section uses JNA[45] to bring the API to JAVA framework.

You need to put the JNA's jar in your classpath and to fix the jna.library.path or java.library.path system property (launch the JVM with -Djna.library.path=...) to point to the libvixAllProducts.so lib, or specify it thanks to LD_LIBRARY_PATH variable on linux or PATH on windows. To deploy your virtual infrastructure with products compatbile with Libvirt[46], you can use a <libvirt> section. Here is an example on how to use it:

[44] http://www.tokeshi.com/index.php?option=com_content&task=view&id=5025
[45] http://jna.java.net/
[46] http://libvirt.org

```
<vms>
<libvirt id="libvirt">
   <hypervisor url="qemu+ssh://jily.activeeon.com/system"/>
   <hypervisor url="xen+ssh://excalibur.activeeon.com"/>
   <hypervisor url="http://bud.activeeon.com"/>
   <authentication user="inria" pwd="inria123"/>
   <image key="myUbuntu" os="unix" />
   <image key="myDebian" os="unix" />
   <image key="myGentoo" os="unix" />
</libvirt>
<vms>
```

The <libvirt> tag says that one want to use the module built on Libvirt API (see the associated section in the previous chapter), it has the following child elements:

```
<hypervisor>
<authentication>
<image>
```

and the following attribute:

```
id="someID"
```

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is Libvirt dependent, see documentation[47] for more information. If you provide a hypervisor without any url or no hypervisor, Libvirt tries to connect locally. You can put how many hypervisors you want.

The <authentication> tag is optional if connecting locally and is used to specify a user with sufficient permissions to manage your virtual environment through Libvirt environment (virt-manager, virsh...). If you connect locally, you may avoid this tag. The only attributes of that section are user and pwd.

The <image> tag points out which virtual machines you want to deploy. It admits the following attributes:

```
key="someName"
os="windows"
```

The “key” attribute designates the unique name of your virtual machine like you registered it (the way it appears thanks to a “virsh -c qemu+ssh://localhost/system list --all”). “os” attribute indicates the guest virtual machine's operating system. Note that that module doesn't support cloning feature.

19.3.6. VirtualBox

This section concerns VirtualBox managed through vboxwebsrv. You need to reference every Virtualizing/src/virtualbox/lib/ jars in the classpath.

[47] http://libvirt.org/uri.html

To deploy your virtual infrastructure with compliant products, you can use a <virtualbox-ws> section (see the associated section in the previous chapter). Here is an example on how to use it:

```
<vms>
  <virtualbox-ws id="virtualbox">
    <hypervisor url="http://psychoquack.activeeon.com:18083"/>
    <hypervisor url="http://noadkoko.activeeon.com:18000"/>
    <authentication user="inria" pwd="inria123"/>
    <image key="myUbuntu" os="unix"/>
    <image key="someLinux" os="unix"/>
  </virtualbox-ws>
<vms>
```

The <virtualbox-ws> tag says that one want to use the module built on VirtualBox SDK (see the associated section in the previous chapter), it has the following child elements:

```
<hypervisor>
<authentication>
<image>
```

and the following attribute:

```
id="someID"
```

The id attribute is made to be able to reference this section from the resource part of the descriptor and is compulsory. The hypervisor element, if specified, indicates the remote hypervisor to connect to. Note that the only attribute (the url) you have to supply is service dependent. It is the url of the vboxwebsrv service you started before launching the deployment.

The <authentication> tag is used to specify a user with sufficient permissions to manager your Virtualbox environment.

The <image> tag points out which virtual machines you want to deploy. It admits the following attributes:

```
key="someName"
os="windows"
```

The “key” attribute designates the unique name of your virtual machine like you registered it. “os” attribute indicates the guest virtual machine's operating system. Note that that module doesn't support cloning feature.

19.4. Troubleshooting.

This section describes the most common errors you can encounter.

- **Libvirt**

```
libvir: Remote error : socket closed unexpectedly
Exception in thread "main" org.ow2.proactive.virtualizing.core.error.VirtualServiceException:
  Unable to connect to qemu+ssh:///system. Unable to connect to libvirt service: qemu+ssh:///
system
```

Be sure that the libvirt daemon is started and to provide a good libvirt url[48]

- **VMware VI/Vix**

```
Exception in thread "main" org.ow2.proactive.virtualizing.core.error.VirtualServiceException:
    Unable to connect to https://localhost:8333/sdk; nested exception is:
    java.net.ConnectException: Connection refused
```

Be sure that the vmware service is started.

```
Exception in thread "main" org.ow2.proactive.virtualizing.core.error.VirtualServiceException: Unable
to connect to http://localhost:8333/sdk: Connection reset
Caused by: java.net.SocketException: Connection reset
```

Your service is up but you provided a bad url. Check protocol and port. By default for VMware server 2 https mapped to port 8333 and http to 8222.

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: Unable to load library 'vixAllProducts':
/usr/lib/vmware-vix/lib/libvixAllProducts.so: wrong ELF class: ELFCLASS64
at com.sun.jna.NativeLibrary.loadLibrary(NativeLibrary.java:127)
```

Use a JRE/JDK that handles the native 32/64bits libraries.

When using vmware-vix to connect to VI compliant software, if you want to get a virtual machine you have to supply the vmx file path from a datastore root, for instance “[standard] myUbuntu/myUbuntu.vmx”, where [standard] is the datacenter's name.

```
- Unable to find the Domain [standard]myUbuntu/myUbuntu.vmx
- mess: The virtual machine cannot be found
Exception in thread "main" org.ow2.proactive.virtualizing.core.error.VirtualServiceException:
    The machine [standard]myUbuntu/myUbuntu.vmx isn't registered for the current host.
```

For the example above, the good path is: “[standard] myUbuntu/myUbuntu.vmx” . Note the space (" ") between the datastore's name and the begin of the path...

[48] http://libvirt.org/uri.html

Chapter 20. Technical Service

20.1. Context

For effective components, non-functional aspects must be added to the application functional code. Likewise enterprise middleware and component platforms, in the context of Grids, services must be deployed at execution in the component containers in order to implement those aspects. This work proposes an architecture for defining, configuring, and deploying such **Technical Services** in a Grid platform.

20.2. Overview

A technical service is a non-functional requirement that may be dynamically fulfilled at runtime by adapting the configuration of selected resources.

From the programmer point of view, a technical service is a class that implements the TechnicalService interface. This class defines how to configure a node.

```
package org.objectweb.proactive.core.descriptor.services;

import java.io.Serializable;
import java.util.Map;

import org.objectweb.proactive.core.node.Node;


/**
 * <p>
 * Interface to implement for defining a Technical Service.
 * </p>
 * <b>Definition of Technical Service:</b>
 * <p>
 * A Technical Service is a non-functional requirement that may be dynamically fulfilled at runtime
 * by updating the configuration of selected resources (here a ProActive Node).
 * </p>
 *
 * @author The ProActive Team
 *
 */
public interface TechnicalService extends Serializable {

    /**
     * Initialize the Technical Service with its argument values.
     *
     * @param argValues
     *            values of the Technical Service arguments.
     */
    public abstract void init(Map<String, String> argValues);

    /**
     * Initialize the given node with the Technical Service.
     *
     * @param node
     *            the node where to apply the Technical Service.
     */
```

```
    public abstract void apply(Node node);
}
```

From the deployer point of view, a technical service is a set of ”variable-value” tuples, each of them configuring a given aspect of the application environment.

```
<technicalServices>
  <class name="myService">
    <property name="name1" value="value1" />
    <property name="name2" value="value2" />
  </class>
</technicalServices>
```

With the previous deployment mechanism, it looks like this:

```
<technical-service id="myService" class="services.Service1">
  <arg name="name1" value="value1" />
  <arg name="name2" value="value2" />
</technical-service>
```

The class attribute defines the implementation of the service. This class has to implement the org.objectweb.proactive.core.descriptor.services.TechnicalService interface.

The configuration parameters of the service are specified by property tags in the descriptor (or by arg tags for the previous version). Those parameters are passed to the init method as a map whose entries correspond to the couples (key = parameter name, value = parameter value). The apply method takes as parameter the node on which the service must be applied. This method is called after the creation or acquisition of a node, and before the node is used by the application.

Note

Two or several technical services could be combined if they touch separate aspects. Indeed, two different technical services, which are conceptually orthogonal, could be **incompatible at source code level** .

That is why a virtual node can be configured by only **one** technical service. However, combining two technical services can be done at source code level, by providing a class extending TechnicalService that defines the correct merging of two concurrent technical services.

20.3. Programming Guide

20.3.1. A full GCM Application Descriptor

```
<GCMApplication xmlns="urn:gcm:application:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">

  <environment>
    <descriptorVariable name="PROACTIVE_LIB" value="dist/lib"/>
    <javaPropertyVariable name="proactive.home" />
    <descriptorDefaultVariable name="NUMBER_OF_VIRTUAL_NODES" value="20"/>
    <programVariable name="VIRTUAL_NODE_NAME"/>
    <javaPropertyVariable name="java.home"/>
    <javaPropertyDescriptorDefault name="host.name" value="localhost"/>
    <javaPropertyProgramDefault name="priority.queue"/>
    <javaPropertyVariable name="user.home" />
```

```
    <descriptorVariable name="hostCapacity" value="2"/>
    <descriptorVariable name="vmCapacity" value="2"/>

    <!-- Include external variables from files-->
    <includePropertyFile location="file.properties"/>
  </environment>


  <application>
    <proactive base="root" relpath="${proactive.home}">
      <configuration>
        <applicationClasspath>
          <!-- Use example -->
          <pathElement base="root" relpath="${proactive.home}/${PROACTIVE_LIB}/ProActive_examples.jar"/
>

          <pathElement base="proactive" relpath="dist/lib/ibis-1.4.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-connect-1.0.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-util-1.0.jar"/>
        </applicationClasspath>
      </configuration>
      <!-- ... -->

      <technicalServices>
        <class name="org.objectweb.proactive.core.body.ft.service.FaultToleranceTechnicalService">
          <property name="global" value="rmi://localhost:1100/FTServer" />
          <property name="resource" value="rmi://localhost:1100/FTServer" />
          <property name="ttc" value="5" />
          <property name="protocol" value="cic" />
        </class>
      </technicalServices>

      <virtualNode id="Workers" capacity="4"/>
  </proactive>
  </application>

  <resources>
    <nodeProvider id="workers">
      <file path="../GCMD_Local.xml" />
    </nodeProvider>
  </resources>
</GCMApplication>
```

20.3.2. A full XML Descriptor File (former deployment)

```
<ProActiveDescriptor>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="master" property="multiple" serviceRefid="ft-master" />
      <virtualNode name="slaves" property="multiple" serviceRefid="ft-slaves" />
    </virtualNodesDefinition>
  </componentDefinition>
  ...
  <infrastructure>
```

```
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess class="JVMNodeProcess" />
      </processDefinition>
    </processes>
  </infrastructure>
  <technicalServiceDefinitions>
    <service id="ft-master" class="services.FaultTolerance">
      <arg name="proto" value="pml" />
      <arg name="server" value="rmi://host/FTServer1" />
      <arg name="TTC" value="60" />
    </service>
    <service id="ft-slaves" class="services.FaultTolerance">
      <arg name="proto" value="cic" />
      <arg name="server" value="rmi://host/FTServer2" />
      <arg name="TTC" value="600" />
    </service>
  </technicalServiceDefinitions>
</ProActiveDescriptor>
```

20.3.3. Nodes Properties

In order to help programmers for implementing their owns technical services, we have added a property system to the nodes. This is useful for configuring technical services.

```
String myProperty = node.getProperty(myKeyAsString);
node.setProperty(myKeyAsString, itsValueAsString);
```

20.4. Further Information

The seminal paper [CDD06c].

The first presentation of this work is available here [1].

The work of this paper [CCDMCompFrame06] is based on technical services.

[1] http://www-sop.inria.fr/oasis/personnel/Alexandre.Di_Costanzo/Alexs_Web_Site/Publications_files/wp4_v1.pdf

Part V. Back matters

Table of Contents

Appendix A. Frequently Asked Questions

Note: This FAQ is under construction. If one of your question is not answered here, just send it at proactive@ow2.org and we will update the FAQ.

Table of Contents

A.1. Frequently Asked Questions

A.1.1. How do I build ProActive from the distribution?

ProActive uses Ant[1] for its build. Assuming that the environment variable JAVA_HOME is properly set to your Java distribution, just go into the compile directory and use the script:

[1] http://jakarta.apache.org/ant/

- on Windows: build.bat deploy.all
- on Unix systems: build deploy.all

'deploy.all' represents the target of the build. This target will compile all sources files and generate the JAR files in the dist/lib directory.

If you want only to compile only parts of ProActive, you should try build, with no arguments. You should see the following output:

```
Buildfile: /home/ffonteno/proactive-git/programming/compile/build.xml

Main targets:

 check.annotations                Perform annotation processing on ProActive code
 check.annotations.all            Do annotation processing for the following ProActive modules : Core + Extensions + Extra
+ Examples + Benchmarks
 check.annotations.benchmarks     Annotation processing for Benchmarks
 check.annotations.core           Annotation processing for Core
 check.annotations.examples       Annotation processing for Examples
 check.annotations.extensions     Annotation processing for Extensions
 check.annotations.extra          Annotation processing for Extra
 clean                          Remove all generated files
 compile                         Deprecated, see compile.all
 compile.all                     Build class files (ProActive + Extensions + Extra + Examples + Utils + Tests)
 compile.benchmarks                Compile the benchmarks
 compile.core                    Compile ProActive core classes
 compile.examples                Compile all examples classes
 compile.extensions              Compile ProActive Extensions classes
 compile.extra                   Compile ProActive Extensions classes
 compile.stubGenerator             Compile Stub Generator Classes
 compile.tests                   Compile functional and unit tests classes
 compile.trywithcatch            Compile ProActive Try With Catch classes
 compile.util                    Compile ProActive Utils classes
 copyright_format                Update Copyright and Version in every text files and format them
 core                           Deprecated, see compile.core
 deploy                          Populate dist/ (with ProActive_examples.jar)
 deploy.all                      Populate dist/ with all ProActive Programming dependencies and JARs
 deploy.annotations.java5          Create a jar file which contains only the JDK 1.5 annotations
 deploy.annotations.java6          Create a jar file which contains only the JDK 1.6 annotations
 deploy.core                     Populate dist/ with all the files needed
 deploy.examples                 Add ProActive_examples.jar to the dist/ directory
 deploy.tests                    Add ProActive_tests.jar to the dist/ directory
 deploy.utils                    Add ProActive_utils.jar to the dist/ directory
 doc.ProActive.doc.zips            Generate the ProActive Programming javadoc and manual zip archives
 doc.ProActive.docs                Generate the ProActive Programming javadoc, manual, and zip archives
 doc.ProActive.javadoc.complete    Generate the ProActive Programming complete javadoc
 doc.ProActive.javadoc.completeZip Generate the ProActive Programming complete javadoc zip
 doc.ProActive.javadoc.published   Generate the ProActive Programming published javadoc
 doc.ProActive.javadoc.publishedZip Generate the ProActive Programming published javadoc zip
 doc.ProActive.manual              Generate all the ProActive Programming manuals  (html, single html, pdf)
 doc.ProActive.manualHtml          Generate the ProActive Programming HTML manual
 doc.ProActive.manualHtmlZip       Generate the ProActive Programming HTML manual zip
 doc.ProActive.manualPdf           Generate the ProActive Programming PDF manual
 doc.ProActive.manualSingleHtml    Generate the ProActive Programming single HTML manual
 doc.ProActive.manualSingleHtmlZip Generate the ProActive Programming single HTML manual zip
 doc.ProActive.schemas             Build documentation for GCM schemas
```

```
examples                        Deprecated, see compile.examples
extensions                      Deprecated, see compile.extensions
extra                           Deprecated, see compile.extra
format                          hibernate's eclipse-based formatter
ibis                            Everything related to ProActive IBIS
jdepend                         JDepend report
junit                           Run all non regression tests on the current host only
junit.clover                    Same as junit but with code coverage enabled
junit.emma                      Same as junit but with code coverage enabled
junit.http                      Run all non regression tests on the current host only
junit.pamr                      Run all non regression tests on the current host only
junit.rmi                       Run all non regression tests on the current host only
junit.rmissh                    Run all non regression tests on the current host only
junit.rmissl                    Run all non regression tests on the current host only
microTimer                      Deprecated, see compile.microTimer
proActiveJar                    Deprecated, see deploy
proActiveWar                    Build the proactive.war archive with CXF embedded
proActiveWarCXF                 Deprecated, see proActiveWar
runPerformanceTest              Run performance test on the current host.
stubGenerator                   Deprecated, see compile.stubGenerator
tutorials                       Extracts tutorials from sources and creates the tutorials directory with building and launching
scripts
update_copyright_and_version    Update Copyright and Version in every text files
uploadSchemas                   upload schemas on web site
Default target: compile.all
```

A.1.2. Why don't the examples and compilation work under Windows?

It happens quite often, that the default installation directory under Windows is under Program Files which contains a space. Then setting the JAVA_HOME environment variable to the install directory, might be a problem for bat files (all windows examples, and compilation are run with bat files). To get rid of those problems, the best thing is to install jdk in a directory whose name does not contain spaces such as C:\java\j2sdk... or D:\java\j2sdk... and then to set the JAVA_HOME environment variable accordingly: set JAVA_HOME=C:\java\j2sdk... Another solution is to do a copy paste of the command defined in the bat file in the DOS window.

A.1.3. Why do I get a Permission denied when trying to launch examples scripts under Linux?

According to the tool used to unpackage the ProActive distribution, permissions of newly created files can be based on default UMASK permissions. If you get a permission denied, just run the command:

```
chmod 755 */*.sh
```

in the ProActive/scripts/unix directory in order to change permissions.

A.1.4. How does the node creation happen?

An active object is always attached to a node. A node represents a logical entity deployed onto a JVM. When creating a new active object, you have to provide an URL or a reference to a node. That node has to exist at the moment you create the active object. It has to be launched on a local or on a remote JVM. In order to be accessible from any remote JVM, a node automatically registers itself in the local RMI Registry on the local machine. Getting a reference to a remote node ends up doing a lookup into a RMI registry. The class NodeFactory provides a method getNode for doing that.

In order to start a node, you can use the startNode[.sh|.bat] script located in the ProActive/bin directory. startNode can only start a node on the local machine. It is not possible to start a remote node using startNode. The reason is that starting a node on a remote host

implies the use of protocol such as RSH, SSH or rLogin that are platform dependant and that cannot be easily abstracted from java. If you want to remotely start a node, you should have a look at Chapter 14, *ProActive Grid Component Model Deployment*.

It is nevertheless possible to create an object on a remote node once it is created. On host X, once you have started a new node using startNode:

```
startNode.sh ///node1
```

you can create an active object from host Y on host X as follows:

```
Node node = NodeFactory.getNode("//X/node1");
// Creates an active object on the remote node
MyClass mc1 = (MyClass) PAActiveObject.newActive(MyClass.class, new Object[] {}, node);

// Turns an existing object into an active object on the remote node
MyClass mc2 = new MyClass();
PAActiveObject.turnActive(mc2, node);
```

You do not need to start any rmi registry manually as they are started automatically.

As we support other ways of registration and discovery (such as HTTP), getting a node can be protocol dependent. For instance, the URL of a node http://host.org/node will not be accessed the same way as rmi://host.org/node. The class NodeFactory is able to read the protocol and to use the right way to access the node.

When an active object is created locally without specifying a node, it is automatically attached to a default node. The default node is created automatically by **ProActive** on the local JVM when a first active object is created without a given node. The name of the default node is generated, based on a random number.

A.1.5. How does the RMI Registry creation happen?

ProActive relies on the RMI Registry for registering and discovering nodes. For this reason, the existence of a RMI Registry is necessary for **ProActive** to be used. In order to simplify the deployment of **ProActive** applications, we have included the creation of the RMI Registry with the creation of nodes. Therefore, if no RMI Registry exists on the local machine, **ProActive** will automatically create one. If one exists, **ProActive** will automatically use it.

A.1.6. What is the class server, why do we need it?

In the RMI model, a class server is a HTTP Server able to answer simple HTTP requests for getting class files. It is needed in the case an object being sent to a remote location where the class the object belongs to is unknown. In such a case, if the property java.rmi.server.codebase has been set properly to an existing class server, RMI will attempt to download the missing class files.

Since **ProActive** makes use of on-the-fly, in memory, generated classes (the stubs), a class server is necessary for each JVM using active objects. For this reason, **ProActive** starts automatically one small class server per JVM. The launching and the use of this class server is transparent to you.

A.1.7. What is a reifiable object?

An object is said to be reifiable if it meets some criteria in order to become an Active Object:

- The object is not of primitive type
- The class of the object is not final
- The object has a constructor with no arguments

A.1.8. What is the body of an active object? What are its local and remote representations?

Once created, an active object is associated with a Body that is the entity managing all the non functional properties of the active object. The body contains the request queue receiving all reified method calls to the reified object (the object from which the active object has been created). It is responsible for storing pending requests and serving them accordingly to a given synchronization policy, which default behavior is FIFO.

The body of the active object should be the only object able to access directly the reified object. All other objects accessing the active object do so through the stub-proxy couple that eventually sends a request to the body. The body owns its own thread that represent the activity of the active object.

The body has two representations. One is local and given by the interface Body:

```
public interface Body extends LocalBodyStrategy, UniversalBody {

  /**
   * Returns whether the body is alive or not. The body is alive as long as it is processing
   * request and reply
   *
   * @return whether the body is alive or not.
   */
  public boolean isAlive();

  /**
   * Returns whether the body is active or not. The body is active as long as it has an associated
   * thread running to serve the requests by calling methods on the active object.
   *
   * @return whether the body is active or not.
   */
  public boolean isActive();

  /**
   * To avoid some causal ordering corruptions, the body can be temporarily set as <i>sterile</i>.
   * Then, it will not be able to send any request, except to himself and to its parent. Such
   * restriction should be necessary when sending multiple requests in parallel.
   *
   * @param isSterile
   * @param parentUID
   * @see org.objectweb.proactive.core.body.proxy.SendingQueueProxy
   */
  public void setSterility(boolean isSterile, UniqueID parentUID);

  /**
   * Get the sterility status of the body
   *
   * @return the sterility status
   * @see org.objectweb.proactive.core.body.proxy.SendingQueueProxy
   */
  public boolean isSterile();

  /**
   * Get the parent UniqueID of the body
   *
   * @return the parent UniqueID
   * @see org.objectweb.proactive.core.body.proxy.SendingQueueProxy
   */
```

```
public UniqueID getParentUID();

/**
 * blocks all incoming communications. After this call, the body cannot receive any request or
 * reply.
 */
public void blockCommunication();

/**
 * Signals the body to accept all incoming communications. This call undo a previous call to
 * blockCommunication.
 */
public void acceptCommunication();

/**
 * Allows the calling thread to enter in the ThreadStore of this body.
 */
public void enterInThreadStore();

/**
 * Allows the calling thread to exit from the ThreadStore of this body.
 */
public void exitFromThreadStore();

/**
 * Tries to find a local version of the body of id uniqueID. If a local version is found it is
 * returned. If not, tries to find the body of id uniqueID in the known body of this body. If a
 * body is found it is returned, else null is returned.
 *
 * @param uniqueID
 *            the id of the body to lookup
 * @return the last known version of the body of id uniqueID or null if not known
 */
public UniversalBody checkNewLocation(UniqueID uniqueID);

/**
 * Returns the MBean associated to this active object.
 *
 * @return the MBean associated to this active object.
 */
public BodyWrapperMBean getMBean();

/**
 * Returns the body that is the target of this shortcut for this component interface
 *
 * @param functionalItfID
 *            the id of the interface on which the shortcut is available
 * @return the body that is the target of this shortcut for this interface
 */
public UniversalBody getShortcutTargetBody(ItfID functionalItfID);

// /**
// * set the policy server of the active object
// * @param server the policy server
```

```
// */
// public void setPolicyServer(PolicyServer server);

/**
 * Set the nodeURL of this body
 *
 * @param newNodeURL
 *          the new URL of the node
 */
public void updateNodeURL(String newNodeURL);

/**
 * For setting an immediate service for this body. An immediate service is a method that will be
 * executed by the calling thread, or by a dedicated per-caller thread if uniqueThread is true.
 *
 * @param methodName
 *          the name of the method
 * @param uniqueThread true if this immediate service should be always executed by the same thread for
 *            a given caller, false if any thread can be used.
 */
public void setImmediateService(String methodName, boolean uniqueThread);

/**
 * For setting an immediate service for this body. An immediate service is a method that will be
 * executed by the calling thread.
 *
 * @param methodName the name of the method
 *
 * @deprecated Replaced by {@link #setImmediateService(String, boolean)}
 */
@Deprecated
public void setImmediateService(String methodName);

/**
 * Removes an immediate service for this body An immediate service is a method that will be
 * executed by the calling thread.
 *
 * @param methodName
 *          the name of the method
 */
public void removeImmediateService(String methodName);

/**
 * Adds an immediate service for this body An immediate service is a method that will be
 * executed by the calling thread, or by a dedicated per-caller thread if uniqueThread is true.
 *
 * @param methodName
 *          the name of the method
 * @param parametersTypes
 *          the types of the parameters of the method
 * @param uniqueThread true if this immediate service should be always executed by the same thread for
 *            a given caller, false if any thread can be used.
 *
 */
```

```
public void setImmediateService(String methodName, Class<?>[] parametersTypes, boolean uniqueThread);

/**
 * Removes an immediate service for this body An immediate service is a method that will be
 * executed by the calling thread.
 *
 * @param methodName
 *          the name of the method
 * @param parametersTypes
 *          the types of the parameters of the method
 */
public void removeImmediateService(String methodName, Class<?>[] parametersTypes);

/**
 * Sets the ForgetOnSend strategy when sending a request <i>methodName</i> to <i>activeObject</i>.
 * @param activeObject
 *            the destination
 * @param methodName
 *            the name of the method
 */
public void setForgetOnSendRequest(Object activeObject, String methodName);

/**
 * Remove the ForgetOnSend setting attached to the given <i>methodName</i> for the given <i>activeObject</i>
 * @param activeObject
 *            the destination
 * @param methodName
 *            the name of the method
 */
public void removeForgetOnSendRequest(Object activeObject, String methodName);

/**
 * Terminate the body. After this call the body is no more alive and no more active. The body is
 * unusable after this call. If some automatic continuations are registered in the futurepool of
 * this body, the ACThread will be killed when the last registered AC is sent.
 */
public void terminate();

/**
 * @param completeACs
 *          if true, this call has the same behavior than terminate(). Otherwise, the ACThread
 *          is killed even if some ACs remain in the futurepool.
 * @see #terminate(). If completeACs is true, this call has the same behavior as terminate().
 *      Otherwise, the ACThread is killed even if some ACs remain in the futurepool.
 */
public void terminate(boolean completeACs);

/**
 * Checks if a method methodName is declared by the reified object AND the method has the same
 * parameters as parametersTypes Note that the called method should be <i>public</i>, since
 * only the public methods can be called on an active object. Note also that a call to
 * checkMethod(methodName, null) is different to a call to checkMethod(methodName, new Class[0])
 * The former means that no checking is done on the parameters, whereas the latter means that we
 * look for a method with no parameters.
```

```
     *
     * @param methodName
     *            the name of the method
     * @param parametersTypes
     *            an array of parameter types
     * @return true if the method exists, false otherwise
     */
    public boolean checkMethod(String methodName, Class<?>[] parametersTypes);

    /**
     * Checks if a method methodName is declared by the reified object Note that the called method
     * should be <i>public</i>, since only the public methods can be called on an active object.
     * Note that this call is strictly equivalent to checkMethod(methodName, null);
     *
     * @param methodName
     *            the name of the method
     * @return true if the method exists, false otherwise
     */
    public boolean checkMethod(String methodName);

    public void registerIncomingFutures();
}
```

This is the local view of the body an object can have when being in the same JVM as the body. For instance, the implementation of the activity of an object done through the method runActivity(Body) of the interface RunActive sees the body locally as it is instantiated by the body itself. The other representation, given by the interface UniversalBody, is remote. It represents the view of the body that a remote object can have and therefore the methods that can be invoked. That view is the one used by the proxy of a remote reference to the active object to send requests to the body of the active object.

```
public interface UniversalBody extends Serializable, SecurityEntity {
    public static Logger bodyLogger = ProActiveLogger.getLogger(Loggers.BODY);
    public static Logger sendReplyExceptionsLogger =
 ProActiveLogger.getLogger(Loggers.EXCEPTIONS_SEND_REPLY);

    /**
     * Receives a request for later processing. The call to this method is non blocking
     * unless the body cannot temporary receive the request.
     * @param request the request to process
     * @exception java.io.IOException if the request cannot be accepted
     * @return value for fault-tolerance protocol
     */
    public int receiveRequest(Request request) throws java.io.IOException, RenegotiateSessionException;

    /**
     * Receives a reply in response to a former request.
     * @param r the reply received
     * @exception java.io.IOException if the reply cannot be accepted
     * @return value for fault-tolerance procotol
     */
    public int receiveReply(Reply r) throws java.io.IOException;

    /**
     * Returns the url of the node this body is associated to
     * The url of the node can change if the active object migrates
```

```
     * @return the url of the node this body is associated to
     */
    public String getNodeURL();

    /**
     * Returns the UniqueID of this body
     * This identifier is unique accross all JVMs
     * @return the UniqueID of this body
     */
    public UniqueID getID();

    /**
     * Return the name of this body, which generally contains the class name of the reified object
     * @return the body name
     */
    public String getName();

    /**
     * Signals to this body that the body identified by id is now to a new
     * remote location. The body given in parameter is a new stub pointing
     * to this new location. This call is a way for a body to signal to his
     * peer that it has migrated to a new location
     * @param id the id of the body
     * @param body the stub to the new location
     * @exception java.io.IOException if a pb occurs during this method call
     */
    public void updateLocation(UniqueID id, UniversalBody body) throws java.io.IOException;

    /**
     * similar to the {@link UniversalBody#updateLocation(org.objectweb.proactive.core.UniqueID, UniversalBody)}
method,
     * it allows direct communication to the target of a functional call, across membranes of composite components.
     * @param shortcut the shortcut to create
     * @exception java.io.IOException if a problem occurs during this method call
     */
    public void createShortcut(Shortcut shortcut) throws java.io.IOException;

    /**
     * Returns the remote friendly version of this body
     * @return the remote friendly version of this body
     */
    public UniversalBody getRemoteAdapter();

    /**
     * Returns the name of the class of the reified object
     * @return the name of the class of the reified object
     */
    public String getReifiedClassName();

    /**
     * Enables automatic continuation mechanism for this body
     * @exception java.io.IOException if a problem occurs during this method call
     */
    public void enableAC() throws java.io.IOException;
```

```
    /**
     * Disables automatic continuation mechanism for this body
     * @exception java.io.IOException if a problem occurs during this method call
     */
    public void disableAC() throws java.io.IOException;

    // FAULT TOLERANCE

    /**
     * For sending a non functional message to the FTManager linked to this object.
     * @param ev the message to send
     * @return depends on the message meaning
     * @exception java.io.IOException if a problem occurs during this method call
     */
    public Object receiveFTMessage(FTMessage ev) throws IOException;

    /**
     * The DGC broadcasting method, called every GarbageCollector.TTB between
     * referenced active objects. The GC{Message,Response} may actually be
     * composed of many GCSimple{Message,Response}.
     *
     * @param toSend the message
     * @return its associated response
     * @throws IOException if a pb occurs during this method call
     */
    public GCResponse receiveGCMessage(GCMessage toSend) throws IOException;

    /**
     * Inform the DGC that an active object is pinned somewhere so cannot
     * be garbage collected until being unregistered.
     * @param registered true for a registration, false for an unregistration
     * @throws IOException if a pb occurs during this method call
     */
    public void setRegistered(boolean registered) throws IOException;

    public String registerByName(String name, boolean rebind) throws IOException, ProActiveException;

    /**
     * @return The main URL of this body (using the default remote object factory)
     */
    public String getUrl();

    /**
     * @return in case of multi protocols, all urls of this body
     */
    public String[] getUrls();

    public String registerByName(String name, boolean rebind, String protocol) throws IOException,
            ProActiveException;
}
```

A.1.9. What is a ProActive stub?

When you create an active object from a regular object, you get in return a reference on an automatically generated **ProActive** stub. **ProActive** uses ASM[2] to generate the stub on the fly. Supposing you have a class A and an instance a of this class, a way to turn the instance a into an active object is to use the method PAActiveObject.turnActive:

```
MyClass mc2 = new MyClass();
PAActiveObject.turnActive(mc2, node);
```

In the code above, the variable a is a direct reference onto the instance of A stored somewhere in memory. In contrast, the variable activeA is a direct reference onto an instance of the generated ProActive stub for the class A. By convention, the ProActive stub of a class A is a class generated in memory by ProActive that inherits from A and that is stored in the package pa.stub as pa.stub.<package of A>.Stub_A. The ProActive stub of a class redefines all public methods to reify them through a generic proxy. The proxy changes all method calls into requests that are sent to the body associated to the reified object (the object pointed by a in our example).

The reified object can be indifferently in the same virtual machine as the active reference or in another one.

A.1.10. Are the call to an active object always asynchronous?

No. Calls to an active object methods are asynchronous under some conditions. This is explained in Section 2.7, “Asynchronous calls and futures”. If, for instance, the return type of a method call is not reifiable, you can use wrappers to keep asynchronism capabilities. Let's suppose that one of your object has such a method:

```
int getNumber();
```

Calling this method with ProActive is synchronous since the 'int' type is not reifiable. To keep asynchronism, it is advised to use the classes given in the org.objectweb.proactive.core.util.wrapper package, or to create your own wrapper based on these examples. In your case, you should use:

```
IntWrapper getNumber();
```

Thus, this new getNumber() method is asynchronous. Remember that **only the methods return type are concerned**, not the parameters.

A.1.11. Why do I get a java.lang.NoClassDefFoundError exception about asm?

ProActive uses ASM[3] for the on-the-fly generation of stub classes. The library asm.jar, provided in the directory Proactive/lib is needed in order to function properly for any active object. If the library is not in the CLASSPATH, you will get the following exception or a similar one:

```
Exception in thread 'main' java.lang.NoClassDefFoundError: org/objectweb/asm/Constants
 at java.lang.ClassLoader.defineClass0(Native Method)
 at java.lang.ClassLoader.defineClass(ClassLoader.java:509)
 at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)
 at java.net.URLClassLoader.defineClass(URLClassLoader.java:246)
 at java.net.URLClassLoader.access$100(URLClassLoader.java:54)
 at java.net.URLClassLoader$1.run(URLClassLoader.java:193)
 at java.security.AccessController.doPrivileged(Native Method)
 at java.net.URLClassLoader.findClass(URLClassLoader.java:186)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
 at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:265)
 at java.lang.ClassLoader.loadClass(ClassLoader.java:262)
 at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:322)
 at org.objectweb.proactive.core.mop.MOP.<clinit>(MOP.java:88)
```

[2] http://asm.objectweb.org/
[3] http://asm.objectweb.org/

```
at org.objectweb.proactive.MOP.createStubObject(MOP.java:836)
at org.objectweb.proactive.MOP.createStubObject(MOP.java:830)
at org.objectweb.proactive.PAActiveObject.newActive(PAActiveObject.java:255)
at org.objectweb.proactive.PAActiveObject.newActive(PAActiveObject.java:180)
at org.objectweb.proactive.examples.binarytree.TreeApplet.main(TreeApplet.java:103)
```

The problem can simply be fixed by adding asm.jar in the CLASSPATH.

A.1.12. Why do I get a java.lang.NoClassDefFoundError exception about bcel?

ProActive also uses BCEL[4] for the on-the-fly generation of stub classes. The library bcel.jar, provided in the directory Proactive/lib is needed in order to function properly for any active object. If the library is not in the CLASSPATH you will get the following exception or a similar one:i

```
Exception in thread 'main' java.lang.NoClassDefFoundError: org/apache/bcel/generic/Type
 at org.objectweb.proactive.core.mop.MOPClassLoader.loadClass(MOPClassLoader.java:129)
 at org.objectweb.proactive.core.mop.MOPClassLoader.loadClass(MOPClassLoader.java:109)
 at org.objectweb.proactive.core.mop.MOP.createStubClass(MOP.java:341)
 at org.objectweb.proactive.core.mop.MOP.findStubConstructor(MOP.java:376)
 at org.objectweb.proactive.core.mop.MOP.createStubObject(MOP.java:443)
 at org.objectweb.proactive.core.mop.MOP.newInstance(MOP.java:165)
 at org.objectweb.proactive.core.mop.MOP.newInstance(MOP.java:137)
 at org.objectweb.proactive.MOP.createStubObject(MOP.java:590)
 at org.objectweb.proactive.MOP.createStubObject(MOP.java:585)
 at org.objectweb.proactive.PAActiveObject.newActive(PAActiveObject.java:170)
 at org.objectweb.proactive.PAActiveObject.newActive(PAActiveObject.java:137)
 at DiscoveryManager.main(DiscoveryManager.java:226)
```

The problem can simply be fixed by adding bcel.jar in the CLASSPATH.

A.1.13. Why do I get a java.security.AccessControlException exception access denied?

If you do not properly set permissions when launching code using ProActive, you may get the following exception or a similar one.

```
java.security.AccessControlException: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
 at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
 at java.security.AccessController.checkPermission(AccessController.java:401)
 at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
 at java.lang.SecurityManager.checkConnect(SecurityManager.java:1044)
 at java.net.Socket.connect(Socket.java:419)
 at java.net.Socket.connect(Socket.java:375)
 at java.net.Socket.<init>(Socket.java:290)
 at java.net.Socket.<init>(Socket.java:118)
 at sun.rmi.transport.proxy.RMIDirectSocketFactory.createSocket(RMIDirectSocketFactory.java:22)
 at sun.rmi.transport.proxy.RMIMasterSocketFactory.createSocket(RMIMasterSocketFactory.java:122)
 at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:562)
 at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:185)
 at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:171)
 at sun.rmi.server.UnicastRef.newCall(UnicastRef.java:313)
 at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
 at org.objectweb.proactive.core.rmi.RegistryHelper.detectRegistry(RegistryHelper.java:101)
 at org.objectweb.proactive.core.rmi.RegistryHelper.getOrCreateRegistry(RegistryHelper.java:114)
```

[4] http://jakarta.apache.org/bcel/

```
at org.objectweb.proactive.core.rmi.RegistryHelper.initializeRegistry(RegistryHelper.java:77)
at org.objectweb.proactive.core.node.rmi.RemoteNodeFactory(RemoteNodeFactory.java:56)
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
at java.lang.reflect.Constructor.newInstance(Constructor.java:274)
at java.lang.Class.newInstance0(Class.java:296)
at java.lang.Class.newInstance(Class.java:249)
at org.objectweb.proactive.core.node.NodeFactory.createNodeFactory(NodeFactory.java:281)
at org.objectweb.proactive.core.node.NodeFactory.createNodeFactory(NodeFactory.java:298)
at org.objectweb.proactive.core.node.NodeFactory.getFactory(NodeFactory.java:308)
at org.objectweb.proactive.core.node.NodeFactory.createNode(NodeFactory.java:179)
at org.objectweb.proactive.core.node.NodeFactory.createNode(NodeFactory.java:158)
...
```

ProActive uses RMI[5] as its underlying transport technology. It uses code downloading features to automatically move generated stub classes from one JVM to another one. For that reason, ProActive needs to install a SecurityManager that controls the execution of the Java code based on a set of permissions given to the JVM. Without explicit permissions nothing is granted for the code running outside java.* or sun.* packages.

See Permissions on JavaTM 2 SDK[6] to learn more about Java permissions.

As a first approximation, in order to run your code, you can create a simple policy file granting all permissions for all code:

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

Then, you need to start your Java program using the property -Djava.security.policy. For instance:

```
java -Djava.security.policy=my.policy.file MyMainClass
```

A.1.14. Why do I get a java.rmi.ConnectException: Connection refused to host: 127.0.0.1 ?

Sometimes, the hosts files (/etc/hosts for UNIX) contains 127.0.0.1 along with the name of the machine. This is an issue for ProActive and JAVA network connections in general. To get round the problem, start your programs with the command line argument:

```
-Dsun.net.spi.nameservice.provider.1=dns,sun"
```

. This tells java not to look at the hosts file, but rather to ask the DNS for network information.

A.1.15. Why aren't my object's fields updated?

Suppose you have a class A with a field a1 as the example below.

```
public class A {
   public void main(String[] args) throws Exception {
      A a = new A();
      A activeA = (A) PAActiveObject.turnActive(a);
      a.a1 = 2; // set the attribute a1 of the instance pointed by a to 2
      activeA.a1 = 2; // set the attribute a1 of the stub instance to 2
   }
}
```

[5] http://java.sun.com/products/jdk/rmi/
[6] http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html

When you reference an active object, you always reference it through its associated stub (see Section 2.6, " Elements of an active object and futures " for the definition of Stub). The stub class inheriting from the reified class, it has also all its fields. But those fields are totally useless as the only role of the generated stub is to reify every public methods call into a request passed to the associated proxy. Therefore accessing directly the fields of an active object through its active reference would result in accessing the fields of the generated stub. This is certainly not the behavior one would expect.

The solution to this problem is very simple: **active object fields should only be accessed through a public method (getter and setter)** . Otherwise, you are accessing the local Stub's fields.

A.1.16. How can I pass a reference on an active object? What is the difference between this and PAActiveObject.getStubOnThis()?

Suppose you have a class A that you want to make active. In A, you want to have a method that returns a reference on that instance of A as the example below.

```
public class A {
   public A getBadRef() {
      return this; // THIS IS WRONG FOR AN ACTIVE OBJECT
   }

}
```

There is indeed a problem in the code above. If an instance of A is created as, or turned into, an active object, the method getRef will in fact be called through the **Body** of the active object by its active thread. The value returned by the method will be the direct reference on the reified object and not a reference on the active object. If the call is issued from another JVM, the value will be passed by copy and the result (assuming A is serializable) will be a deep copy of A with no links to the active object.

The solution, if you want to pass a link to the active object from the code of the reified object, is to use the method PAActiveObject.getStubOnThis(). This method will return the reference to the stub associated to the active object whose thread is calling the method. The correct version of the previous class is:

```
public class A {
   public A getGoodRef() {
      return (A) PAActiveObject.getStubOnThis(); // returns a reference on the stub
   }

}
```

A.1.17. How can I create an active object?

To create an active object, you can invoke one of the methods newActive or turnActive of the ProActive class. PAActiveObject.newActive creates an active object based on the instantiation of a new object whereas PAActiveObject.turnActive creates an active object based on an existing object. Both of the newActive or turnActive methods allow you to specify where to create the active object (which node) and to customize its activity or its body (see Q: A.1.18).

Here is a simple example creating an active object of class A in the local JVM. If the invocation of the constructor of class A throws an exception, it is placed inside an exception of type ActiveObjectCreationException. When the call to newActive returns, the active object has been created and its active thread is started.

```
public class A {
```

```
  private int i;
  private String s;

  public A() {
  }

  public A(int i, String s) {
     this.i = i;
     this.s = s;
  }


  public void main(String[] args) throws Exception {

     // instance based creation
     A a1;
     Object[] params = new Object[] { new Integer(26), "astring" };
     try {
        a = (A) PAActiveObject.newActive(A.class.getName(), params);
     } catch (ActiveObjectCreationException e) {
        // creation of ActiveObject failed
        e.printStackTrace();
     }
     // object based creation
     A a2 = new A(26, "astring");
     try {
        a = (A) PAActiveObject.turnActive(a);
     } catch (ActiveObjectCreationException e) {
        // creation of ActiveObject failed
        e.printStackTrace();
     }
  }
}
```

A.1.18. What are the differences between instantiation based and object based active objects creation?

In **ProActive**, there are two ways to create active objects. One way is to use PAActiveObject.newActive and is based on the instantiation of a new object, the other is to use PAActiveObject.turnActive and is based on the use of an existing object.

When using instantiation based creation, any argument passed to the constructor of the reified object through PAActiveObject.newActive is serialized and passed by copy to the object. This is because the model behind **ProActive** is uniform whether the active object is instantiated locally or remotely. The parameters are therefore guaranteed to be passed by copy to the constructor. When using PAActiveObject.newActive, you have to make sure that the arguments of the constructor are Serializable. On the other hand, the class used to create the active object **does not need to be**Serializable even in case the active object is created remotely.

When using object based creation, you create the object that is going to be reified as an active object before hand. Therefore there is no serialization involved when you create the object. When you invoke PAActiveObject.turnActive on the object two cases are possible. If you create the active object locally (on a local node), it will not be serialized. If you create the active object remotely (on a remote node), the reified object will be serialized. Therefore, if the turnActive is done on a remote node, the class used to create the active object **has to be**Serializable. In addition, when using turnActive, care must be taken that no other references to the originating object are kept by other objects after the call to turnActive. A direct call to a method of the originating object without passing by a ProActive stub on this object will break the model.

A.1.19. Why do I have to write a no-args constructor?

ProActive automatically creates a stub/proxy pair for your active objects. A constructor with no-args needs to be used for active object creation since the constructor is called for the stub creation. Avoid placing initialization code in this constructor, as it may lead to unexpected behavior.

A.1.20. How do I control the activity of an active object?

As explained in Section 2.5.5, "Using A Custom Activity", there are two ways to define the activity of your active object

- Implementing one or more of the sub-interfaces of Active directly in the class used to create the active object
- Passing an object implementing one or more of the sub-interfaces of Active in parameter to the method newActive or turnActive

Implementing the interfaces directly in the class used to create the active object

This is the easiest solution when you do control the class that you make active. Depending on which phase in the life of the active object you want to customize, you implement the corresponding interface (one or more) amongst InitActive, RunActive and EndActive. Here is an example that has a custom initialization and activity.

```
public class B implements InitActive, RunActive {

   private String myName;

   public String getName() {
      return myName;
   }

   // -- implements InitActive
   public void initActivity(Body body) {
      myName = body.getName();
   }

   // -- implements RunActive for serving request in a LIFO fashion
   public void runActivity(Body body) {
      Service service = new Service(body);
      while (body.isActive()) {
         service.blockingServeYoungest();
      }
   }

   public void main(String[] args) throws Exception {
      B b = (B) PAActiveObject.newActive(B.class.getName(), null);
      System.out.println("Name = " + b.getName());
   }
}
```

Passing an object implementing the interfaces when creating the active object

This is the solution to use when you do not control the class that you make active or when you want to write generic activity policy and reused them with several active objects. Depending on which phase in the life of the active object you want to customize, you implement the corresponding interface (one or more) amongst InitActive, RunActive and EndActive. Here is an example that has a custom activity.

Compared to the solution above where interfaces are directly implemented in the reified class, there is one restriction here: you cannot access the internal state of the reified object. Using an external object should therefore be used when the implementation of the activity is generic enough not to have to access the member variables of the reified object.

```
public class LIFOActivity implements RunActive {

   // -- implements RunActive for serving request in a LIFO fashion
   public void runActivity(Body body) {
      Service service = new Service(body);
      while (body.isActive()) {
         service.blockingServeYoungest();
      }
   }
}

public class C implements InitActive {
   private String myName;

   public String getName() {
      return myName;
   }

   // -- implements InitActive
   public void initActivity(Body body) {
      myName = body.getName();
   }

   public void main(String[] args) throws Exception {
      C c = (C) PAActiveObject.newActive(C.class, null, null, null, new LIFOActivity(), null);
      System.out.println("Name = " + c.getName());
   }
}
```

A.1.21. What happened to the former live() method and Active interface?

The former Active interface was simply a marker interface allowing to change the body and/or the proxy of an active object. It was useless most of the time and was made obsolete with the introduction of the MetaObjectFactory in the **0.9.3** release.

```
public interface MetaObjectFactory {

   /**
    * Creates or reuses a RequestFactory
    * @return a new or existing RequestFactory
    * @see RequestFactory
    */
   public RequestFactory newRequestFactory();

   /**
    * Creates or reuses a ReplyReceiverFactory
    * @return a new or existing ReplyReceiverFactory
    * @see ReplyReceiverFactory
    */
   public ReplyReceiverFactory newReplyReceiverFactory();

   /**
    * Creates or reuses a RequestReceiverFactory
    * @return a new or existing RequestReceiverFactory
```

```
     * @see RequestReceiverFactory
     */
    public RequestReceiverFactory newRequestReceiverFactory();

    /**
     * Creates or reuses a RequestQueueFactory
     * @return a new or existing RequestQueueFactory
     * @see RequestQueueFactory
     */
    public RequestQueueFactory newRequestQueueFactory();

    /**
     * Creates or reuses a MigrationManagerFactory
     * @return a new or existing MigrationManagerFactory
     * @see MigrationManagerFactory
     */
    public MigrationManagerFactory newMigrationManagerFactory();

    //    /**
    //     * Creates or reuses a RemoteBodyFactory
    //     * @return a new or existing RemoteBodyFactory
    //     * @see RemoteBodyFactory
    //     */
    //    public RemoteBodyFactory newRemoteBodyFactory();

    /**
     * Creates or reuses a ThreadStoreFactory
     * @return a new or existing ThreadStoreFactory
     * @see ThreadStoreFactory
     */
    public ThreadStoreFactory newThreadStoreFactory();

    // GROUP

    /**
     * Creates or reuses a ProActiveGroupManagerFactory
     * @return a new ProActiveGroupManagerFactory
     */
    public ProActiveSPMDGroupManagerFactory newProActiveSPMDGroupManagerFactory();

    /**
     * creates a ProActiveComponentFactory
     * @return a new ProActiveComponentFactory
     */

    // COMPONENTS
    public PAComponentFactory newComponentFactory();

    /** Creates a DebuggerFactory
     * @return a new DebuggerFactory
     */
    public DebuggerFactory newDebuggerFactory();

    /**
```

```
     * accessor to the parameters of the factory (object-based configurations)
     * @return the parameters of the factory
     */

    // COMPONENTS
    public Map<String, Object> getParameters();

    //SECURITY

    /**
     * Creates the ProActiveSecurityManager
     * @return a new ProActiveSecurityManager
     * @see ProActiveSecurityManager
     */
    public ProActiveSecurityManager getProActiveSecurityManager();

    public void setProActiveSecurityManager(ProActiveSecurityManager psm);

    public Object clone() throws CloneNotSupportedException;

    // REQUEST-TAGS
    /**
     * Create the RequestTags manager.
     * @return the RequestTags manager
     */
    public MessageTagsFactory newRequestTagsFactory();

    // FAULT-TOLERANCE

    /**
     * Creates the fault-tolerance manager.
     * @return the fault-tolerance manager.
     */
    public FTManagerFactory newFTManagerFactory();

    // TIMING

    /**
     * A setter for the reductor.
     * @param timitReductor
     */
    public void setTimItReductor(Object timItReductor);

    /**
     * A getter for the reductor stub.
     * @return The stub on the reductor
     */
    public Object getTimItReductor();
}
```

Up to **ProActive 0.9.3**, the activity of an active object was given by a method live(Body) called by reflection of the reified object. Doing this way did not allow compile time type checking of the method, did not allow to externalize from the reified object its activity and did not allow to give a custom activity to an active object created using turnActive. We solved all those issues using the new mechanism based on the three interfaces InitActive, RunActive and EndActive.

In order to convert the code of an active object containing a method live to the new interface you just need to:

- implement the new interface RunActive (and remove Active if it was implemented)
- changed the name of the method live to runActivity

A.1.22. Why should I avoid to return null in methods body?

On the **caller** side, the test:

```
if (result_from_method == null)
```

has no sense. Indeed, result_from_method is a couple Stub-FutureProxy as explained above, so even if the method returns null, result_from_method cannot be null:

```
public MyClass getMyClass(boolean returnNull) {
   if (!returnNull) {
      return new MyClass();
   } else {
      return null; //--> to avoid in ProActive
   }
}
```

On the caller side:

```
MyClass o = (MyClass) PAActiveObject.newActive(MyClass.class, null);
MyClass result_from_method = o.getMyClass(true);
if (result_from_method == null) {
   // Do something
}
```

This test is never true. Indeed, result_from_method is either **Stub-->Proxy-->null** if the future is not yet available or if the method returns null, or it is **Stub-->Proxy-->Object** if the future is available, but result_from_method is **never null**. See Documentation on Futures in Section 2.7.4, “Good ProActive programming practices” for more information about common errors to avoid.

A.1.23. How do I make a Component version out of an Active Object version?

There is such an example, in the ProActive/examples/components/c3d directory. The code for c3d is adapted to use components.

There are several steps to cover:

1. Make sure you have made interfaces for the objects which are to be turned into components. This is needed to be able to do the binding between components.
2. Replace the references to Active Object classes by their interfaces.
3. Create a component wrapper for each Active Object which should appear as a component. It should contain the binding behavior (bindFc,unbindFc,listFc,lookupFc methods), and maybe handle field modification.
4. Create a main class where the components are created then bound, or use an ADL file to do so.

A.1.24. Why is my call not asynchronous?

ProActive allows to have asynchronous code, in the following cases:

- The return value is reifiable (see Q: A.1.7). This is needed to ensure the creation of the Future, which is the returned container (the future is used while waiting for the effective result to arrive). The returned class has to be Serializable, can not be final, and must have an empty no-arguments constructor.

- The return value is void. In this case, the rendezvous is made, and then the caller resumes its activity, while the receiver has now a new methodCall in its queue.

More explanations can be found in Section 2.7, “Asynchronous calls and futures”.

A.1.25. What is the difference between passing parameters in deployment descriptors and setting properties in ProActive Configuration file?

Parameters defined in deployment descriptor should be only JVM related, whereas properties set in the configuration file are ProActive properties or user-defined properties. They are used with a different approach: parameters given in descriptors are part of the java command that will create other JVMs, whereas properties will be loaded once JVMs are created.

A.1.26. Why ProActive is slow to start ?

Since ProActive Programming 4.1.0, an embedded Jetty HTTP server is started. By default, Jetty uses the java.security.SecureRandom random number generator. It uses the operating system's source to provide entropy. If your machine is very calm, then there may not be enough entropy to drive the random number generator and hence the operating system waits for more interrupts, disk IO, network traffic or whatever is used to generate the entropy.

See the Jetty documentation[7] for more details.

A.1.27. Why do I see “The ProActive log4j collector is still not available, printing logging events on the console to avoid log loss” when a ProActive runtime is started ?

Since ProActive Programming 4.1.0, a distributed log4j mechanism is enabled by default when an application is started with the GCM deployment framework (see Section 11.6, “Log4j configuration”).

This mechanism expects to be able to contact a log collector located on the machine who loaded the GCM Application Descriptor. If a started ProActive runtime is not able to contact the log collector within 30 seconds then all the logging events are printed on the standard output to avoid data loss and help the troubleshooting. As soon as the log collector is available all the printed logging entries are sent to the log collector.

Q: A.1.26 can cause a such message.

A.1.28. About the former deployment process: why do I get the following message when parsing my XML deployment file: ERROR: file:~/ProActive/descriptor.xml Line:2 Message:cvc-elt.1: Cannot find the declaration of element 'ProActiveDescriptor'?

This message turns up because the Schema cannot be found. Indeed, at the beginning of our XML deployment files, we put the line:

```
<ProActiveDescriptor xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:noNamespaceSchemaLocation='DescriptorSchema.xsd'>
```

which means, the schema named DescriptorSchema.xsd is expected to be found in the current directory to validate the XML. Be sure you have this file in the same directory than your file, or just change the path to point to the correct schema.

[7] http://docs.codehaus.org/display/JETTY/Connectors+slow+to+startup

Appendix B. Reference Card

ProActive is a Java library for **parallel**, **distributed**, and **concurrent** computing, also featuring **mobility** and **security** in a uniform framework. **ProActive** provides a comprehensive API and a graphical interface. The library is based on an Active Object pattern that is a uniform way to encapsulate:

- **a remotely** accessible object,
- a **thread** as an asynchronous activity,
- an **actor** with its own script,
- a **server** of incoming requests,
- a **mobile** and potentially secure entity,
- a **component** with server and client interfaces.

ProActive is only made of standard Java classes, and requires **no changes to the Java Virtual Machine**. Overall, it simplifies the programming of applications distributed over Local Area Network (LAN), Clusters, Intranet or Internet GRIDs.

B.1. Main concepts and definitions

- **Active Objects (AO):** a remote object, with its own thread, receiving calls on its public methods
- **FIFO activity:** an AO, by default, executes the request it receives one after the other, in the order they were received
- **No-sharing:** standard Java objects cannot be referenced from 2 AOs, ensured by deep-copy of constructor params, method params, and results
- **Asynchronous Communications:** method calls towards AOs are asynchronous
- **Future:** the result of a non-void asynchronous method call
- **Request:** the occurrence of a method call towards an AO
- **Service:** the execution by an AO of a request
- **Reply:** after a service, the method result is sent back to the caller
- **Wait-by-necessity:** automatic wait upon the use of a still awaited future
- **Automatic Continuation:** transmission of futures and replies between AO and JVMs
- **Migration:** an AO moving from one JVM to another, computational weak mobility: the AO decides to migrate and stack is lost
- **Group:** a typed group of objects or AOs. Methods are called in parallel on all group members.
- **Component:** made of AOs, a component defines server and client interfaces
- **Primitive Component:** directly made of Java code and AOs
- **Composite Component:** contains other components (primitives or composites)
- **Parallel Component:** a composite that is using groups to multicast calls to inner components
- **Security:** X.509 Authentication, Integrity, and Confidentiality defined at deployment in an XML file on entities such as communications, migration, dynamic code loading.
- **Virtual Node (VN):** an abstraction (a string) representing where to locate AOs at creation
- **Deployment descriptor:** an XML file where a mapping VN --> JVMs --> Machine is specified.
- **Node:** the result of mapping a VN to a set of JVMs. After activation, a VN contains a set of nodes, living in a set of JVMs.

- **IC2D:** Interactive Control and Debugging of Distribution: a Graphical environment for monitoring and steering Grid applications

B.2. Main Principles: Asynchronous Method Calls And Implicit futures

```
// Create an active Object of type A in the JVM specified by Node
A a = (A) PAActiveObject.newActive('A', params, node);

// A one way typed asynchronous communication towards the (remote) active object a
// A request is sent to a
a.foo (param);

// A typed asynchronous communication with result.
// v is first an awaited Future to be transparently filled up after
// service of the request and reply
v = a.bar (param);

...

// Use of the result of an asynchronous call.
// If v is still an awaited future, it triggers an automatic
// wait-by-necessity
v.gee (param);
```

B.3. Explicit Synchronization

```
// Returns True if the object is still an awaited Future
boolean PAFuture.isAwaited(Object);

// Blocks until the object is no longer awaited
void PAFuture.waitFor(Object);

// Blocks until all the objects in Vector are no longer awaited
void PAFuture.waitForAll(Vector);

// Blocks until one of the objects in Vector is no longer awaited
// Returns the index of the available future
int PAFuture.waitForAny(Vector);
```

B.4. Programming Active Objects' Activity And Services

When an AO must implement an activity that is not FIFO, the RunActive interface has to be implemented: it specifies the AO behavior in the method named runActivity():

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
```

```
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
package org.objectweb.proactive;

import org.objectweb.proactive.annotation.PublicAPI;


/**
 * <P>
 * RunActive is related to the activity of an active object.
 * When an active object is started, which means that its
 * active thread starts and serves the requests being sent
 * to its request queue, it is possible to define exactly how
 * the activity (the serving of requests amongst others) will
 * be done.
 * </P><P>
 * An object implementing this interface is invoked to run the
 * activity until an event trigger its end. The object being
 * reified as an active object can directly implement this interface
 * or an external class can also be used.
 * </P>
 * <P>
 * It is the role of the body of the active object to perform the
 * call on the object implementing this interface. For an active object
 * to run an activity, the method <code>runActivity</code> must not end
 * before the end of the activity. When the method <code>runActivity</code>
 * ends, the activity ends too and the <code>endActivity</code> can be invoked.
 * </P>
 * <P>
 * Here is an example of a simple implementation of <code>runActivity</code> method
 * doing a FIFO service of the request queue :
```

```
 * </P>
 * <pre>
 * public void runActivity(Body body) {
 *   Service service = new Service(body);
 *   while (body.isActive()) {
 *     service.blockingServeOldest();
 *   }
 * }
 * </pre>
 *
 * @author The ProActive Team
 * @version 1.0,  2002/06
 * @since   ProActive 0.9.3
 */
@PublicAPI
public interface RunActive extends Active {

    /**
     * Runs the activity of the active object.
     * @param body the body of the active object being started
     */
    public void runActivity(Body body);
}
```

Example:

```
// Implements RunActive for programming a specific behavior
public class A implements RunActive {
  // runActivity() is automatically called when such an AO is created
  public void runActivity(Body body) {
    Service service = new Service(body);
    while (terminate) {
      ... // Do some activity on its own
      ...
      ... // Do some services, e.g. a FIFO service on method named foo
      service.serveOldest('foo');
      ...
    }
  }
}
```

Two other interfaces can also be specified:

The method `initActivity(Body body)` initializes the activity of the active object. It is called once before the `runActivity(Body body)` method. It is not called again if the active object restarts after migration.

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
```

```
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
package org.objectweb.proactive;

import org.objectweb.proactive.annotation.PublicAPI;


/**
 * <P>
 * InitActive is related to the initialization of the activity of an
 * active object. The initialization of the activity is done only once.
 * In case of a migration, an active object restarts its activity
 * automatically without reinitializing.
 * </P><P>
 * An object implementing this interface can be invoked to perform the
 * initialization work before the activity is started. The object being
 * reified as an active object can implement this interface or an external
 * class can also be used.
 * </P>
 * <P>
 * It is generally the role of the body of the active object to perform the
 * call on the object implementing this interface.
 * </P>
 *
 * @author The ProActive Team
 * @version 1.0,  2002/06
 * @since   ProActive 0.9.3
 */
@PublicAPI
```

```
public interface InitActive extends Active {

   /**
    * Initializes the activity of the active object.
    * @param body the body of the active object being initialized
    */
   public void initActivity(Body body);
}
```

The method endActivity(Body body) finalizes the activity of the active object. It is called once after the execution of the runActivity(Body body) method.

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
package org.objectweb.proactive;

import org.objectweb.proactive.annotation.PublicAPI;


/**
```

```
 * <P>
 * EndActive is related to the finalization of an active object.
 * The finalization of the activity is done only once when the object
 * stops to be active and becomes unusable.
 * In case of a migration, an active object stops its activity
 * before restarting on a new VM automatically without finalization.
 * </P><P>
 * An object implementing this interface can be invoked to perform the
 * finalization work after the activity is ended. The object being
 * reified as an active object can implement this interface or an external
 * class can also be used.
 * </P>
 * <P>
 * It is generally the role of the body of the active object to perform the
 * call on the object implementing this interface.
 * </P>
 * <P>
 * It is hard to ensure that the <code>endActivity</code> method will indeed
 * be invoked at the end of the activity. <code>Error<code>, <code>Exception<code>,
 * customized activity that never ends or sudden death of the JVM can prevents
 * this method to be called by the body of the active object.
 * </P>
 *
 * @author The ProActive Team
 * @version 1.0,  2002/06
 * @since   ProActive 0.9.3
 */
@PublicAPI
public interface EndActive extends Active {

    /**
     * Finalized the active object after the activity has been stopped.
     * @param body the body of the active object being finalized.
     */
    public void endActivity(Body body);
}
```

B.5. Reactive Active Object

Even when an AO is busy doing its own work, it can remain reactive to external events (method calls). One just has to program non-blocking services to take into account external inputs.

```
public class BusyButReactive implements RunActive {

    public void  runActivity(Body body) {
        Service service = new Service(body);
        while ( ! hasToTerminate ) {
            ...
            // Do some activity on its own ...
            ...
            // Non blocking service ...
            service.serveOldest('changeParameters', 'terminate');
            ...
            }
```

```
    }

  public void  changeParameters () {
        .....
        // change computation parameters
        }

  public void  terminate (){
        hasToTerminate=true;
        }
  }
```

It also allows one to specify explicit termination of AOs. Of course, the reactivity is up to the length of going around the loop. Similar techniques can be used to start, suspend, restart, and stop AOs.

B.6. Service methods

The following method are in the Service class. They can be used to control how requests are served.

Non-blocking services: returns immediately if no matching request is pending

```
// Serves the oldest request in the request queue <emphasis role="bold">
void serveOldest();

// Serves the oldest request aimed at a method of name methodName
void serveOldest(String methodName)

// Serves the oldest request matching the criteria given be the filter
void serveOldest(RequestFilter requestFilter)
```

Blocking services: waits until a matching request can be served

```
// Serves the oldest request in the request queue
void blockingServeOldest();

//Serves the oldest request aimed at a method of name methodName
void blockingServeOldest(String methodName)

// Serves the oldest request matching the criteria given be the filter
void blockingServeOldest(RequestFilter requestFilter)
```

Blocking timed services: wait a matching request at most a time given in ms

```
// Serves the oldest request in the request queue.
// Returns after timeout (in ms) if no request is available
void blockingServeOldest (long timeout)

// Serves the oldest request aimed at a method of name methodName
// Returns after timeout (in ms) if no request is available
void blockingServeOldest(String methodName, long timeout)

// Serves the oldest request matching the criteria given by the filter
void blockingServeOldest(RequestFilter requestFilter)
```

Waiting primitives:

```
// Wait until a request is available or until the body terminates
void waitForRequest();

// Wait until a request is available on the given method name,
// or until the body terminates
void  waitForRequest(String methodName);
```

Others:

```
// Start a FIFO service policy. Call does not return. In case of
// a migration, a new runActivity() will be started on the new site
void fifoServing();

// Invoke a LIFO policy. Call does not return. In case of
// a migration, a new runActivity() will be started on the new site
void lifoServing()

// Serves the youngest request in the request queue
void serveYoungest()

// Removes all requests in the pending queue
void flushAll()
```

B.7. Active Object Creation:

```
// Creates a new AO of type classname. The AO is located on the given node,
// or on a default node in the local JVM if the given node is null
Object PAActiveObject.newActive(String classname, Object[] constructorParameters, Node node);

// Creates a new set of AO of type classname.
// The AO are located on each JVMs the Virtual Node is mapped onto
Object PAActiveObject.newActive(String classname,Object[] constructorParameters,VirtualNode virtualnode);

// Copy an existing Java object and turns it into an AO.
// The AO is located on the given node, or on a default node in
Object turnActive(Object, Node node);
```

B.8. Groups:

```
// Created a group of AO of type 'A' in the JVMs specified
// by nodes. ga is a Typed Group of type 'A'.
// The number of AO being created matches the number of param arrays.
// Nodes can be a Virtual Node defined in an XML descriptor */
A ga = (A) PAGroup.newGroup( 'A', params, nodes);

// A general group communication without result.
// A request to foo is sent in parallel to the active objects in the in group ga  */
ga.foo(...);

// A general group communication with a result.
// gv is a typed group of 'V', which is first a group
// of awaited Futures, to be filled up asynchronously
V gv = ga.bar(...);
```

```
// Use of the result of an asynchronous group call. It is also a
// collective operation: gee method is called in parallel on each object in  the group.
// Wait-by-necessity occurs when the results are awaited
gv.gee (...);

// Get the group representation of a typed group
Group ag = PAGroup.getGroup(ga);

// Add object in the group ag.
// o can be a standard Java object or an AO,
// it must be of a compatible type
ag.add(o);

// Removes the object at the specified index
ag.remove(index)

// Returns to the typed view of a group
A ga2 = (A) ag.getGroupByType();

// By default, a group used as a parameter of a group communication
// is sent as a deep copy of the group.
// On a group call (ga.foo(g)) the scatter
// parameter is dispatched in a round robin fashion to the active objects in the
// target group, e.g. upon ga.foo(g)
void PAGroup.setScatterGroup(g);

// Get back to the default: entire group transmission in all group
// communications, e.g. upon ga.foo(g)
void PAGroup.unsetScatterGroup(g);
```

B.9. Explicit Group Synchronizations

Methods both in Interface Group, and static in class PAGroup

```
// Returns True if object is a group and all members are still awaited
boolean PAGroup.allAwaited (Object);

// Returns False only if at least one member is still awaited
boolean PAGroup.allArrived (Object);

// Wait for all the members in group to arrive (all no longer awaited)
void PAGroup.waitAll (Object);

// Wait for at least nb members in group to arrive
void PAGroup.waitN (Object, int nb);

// Waits for at least one member to arrived, and returns its index
int PAGroup.waitOneAndGetIndex (Object);
```

B.10. OO SPMD

```
// Creates an SPMD group and creates all members with params on the nodes.
// An SPMD group is a typed group in which every member has a reference to
// the others (the SPMD group itself).
```

```
A spmdGroup  =  (A) PASPMD.newSPMDGroup('A', params, nodes);

// Returns the SPMD group of the activity.
A mySpmdGroup = (A) PASPMD.getSPMDGroup();

// Returns the rank of the activity in its SPMD group.
int rank = PASPMD.getMyRank();

// Blocks the activity (after the end of the current service) until all
// other members of the SPMD group invoke the same barrier.
// Three barriers are available: total barrier, neighbors based barrier
// and method based barrier.
PASPMD.barrier('barrierID');
```

B.11. Migration

Methods both in the interface MigrationController, and in class PAMobileAgent

```
// Migrate the current AO to the same JVM as the AO
void migrateTo(Object o);

// Migrate the current AO to JVM given by the node URL
void migrateTo(String nodeURL);

// Migrate the current AO to JVM given by the node
int void migrateTo(Node node);
```

To initiate the migration of an object from outside, define a public method, that upon service will call the static migrateTo primitive:

```
public void moveTo(Object) {
  try {
    PAMobileAgent.migrateTo(t);
  }
  catch (Exception e) {
    e.printStackTrace();
    logger.info('Cannot migrate.');
  }
}
```

To define a migration strategy we implement the methods in in the interface MigrationStrategyManager:

```
// Specification of a method to execute before migration <emphasis role="bold">
void onDeparture(String MethodName);

// Specification of a method to execute after migration, upon the
// arrival in a new JVM
void onArrival(String MethodName);

// Specifies a migration itinerary <emphasis role="bold">
void setMigrationStrategy(MigrationStrategy);

// Adds a JVM destination to an itinerary <emphasis role="bold">
void add(Destination);

// Remove a JVM destination in an itinerary
```

```
void remove(Destination d);
```

B.12. Components

Components are formed from AOs. A component can be linked and can communicate with other remote components. A component can be composite, made of other components, and distributed over several machines. Component systems are defined in XML files (ADL: Architecture Description Language); these files describe the definition, the assembly, and the bindings of components.

Components follow the Fractal hierarchical component model[1] specification and API.

The following methods are specific to ProActive.

In the class PAComponent:

```
// Creates a new ProActive component from the specified class A.
// The component is distributed on JVMs specified by the Virtual Node
// The ComponentParameters defines the configuration of a component:
// name of component, interfaces (server and client), etc.
// Returns a reference to a component, as defined in the Fractal API
Component PAComponent.newActiveComponent('A', params, VirtualNode, ComponentParameters);
```

Fractive:

```
// This method is used in primitive components.
// It generates a client collective interface named itfName, and typed as itfSignature.
// This collective interface is a typed ProActive group.
ProActiveInterface createCollectiveClientInterface(String itfName, String itfSignature);
```

B.13. Security:

ProActive has an X.509 Public Key Infrastructure (PKI) allowing communication Authentication, Integrity, and Confidentiality (AIC) to be configured in an XML security file at deployment and outside any source code. Security is compatible with mobility, allows for hierarchical domain specification and dynamically negotiated policies.

Example of specification:

```
<Rule>
  <From>
   <Entity type='VN' name='VN1'/>
  </From>
  <To>
    <Entity type='VN' name='VN2'/>
  </To>
  <Communication>
    <Request value='authorized'>
      <Attributes authentication='required' integrity='required' confidentiality='optional'/>
    </Request>
  </Communication>
  <Migration>denied</Migration>
  <AOCreation>denied</AOCreation>
</Rule>
```

This rule specifies that communications (requests) are authorized from Virtual Node 'VN1' to the VN 'VN2', provided that authentication and integrity are being used, while confidentiality is optional. Migration and AO creation are not authorized.

[1] http://fractal.objectweb.org

B.14. Deployment

Virtual Nodes (VN) allow one to specify the location where to create AOs. A VN is uniquely identified as a String, is defined in an XML Deployment Descriptor where it is mapped onto JVMs. JVMs are themselves mapped onto physical machines: VN --> JVMs --> Machine. Various protocols can be specified to create JVMs onto machines (ssh, Globus, LSF, PBS, rsh, rlogin, Web Services, etc.). After activation, a VN contains a set of nodes, living in a set of JVMs. Overall, VNs and deployment descriptors allow to abstract away from source code: machines, creation, lookup and registry protocols.

Example of a deployment descriptor that provides the localhost as resource with a capacity of hostCapacity jvms, each jvm hosting vmCapacity nodes.

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

  <environment>
    <javaPropertyVariable name="user.home" />

    <javaPropertyDescriptorDefault name="os" value="windows" />
  </environment>

  <resources>
    <host refid="hLocalhost" />
  </resources>

  <infrastructure>

    <hosts>
      <host id="hLocalhost" os="${os}" hostCapacity="${hostCapacity}" vmCapacity="${vmCapacity}">
        <homeDirectory base="root" relpath="${user.home}" />
      </host>

    </hosts>

  </infrastructure>
</GCMDeployment>
```

An example of application deployment:

```
<?xml version="1.0" encoding="UTF-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd">

  <environment>
    <javaPropertyVariable name="proactive.home" />
    <javaPropertyVariable name="user.home" />

    <descriptorVariable name="hostCapacity" value="4"/>
    <descriptorVariable name="vmCapacity" value="1"/>
  </environment>
```

```
  <application>
    <proactive base="root" relpath="${proactive.home}">
      <configuration>
        <applicationClasspath>
          <pathElement base="proactive" relpath="dist/lib/ProActive_examples.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-1.4.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-connect-1.0.jar"/>
          <pathElement base="proactive" relpath="dist/lib/ibis-util-1.0.jar"/>
        </applicationClasspath>
      </configuration>

      <virtualNode id="Agent" capacity="4"/>

    </proactive>
  </application>

  <resources>
    <nodeProvider id="LOCAL">
      <file path="../GCMD_Local.xml" />
    </nodeProvider>
  </resources>

</GCMApplication>
```

Deployment API

Deployment is done in several steps: load the XML descriptor file, activate the virtual nodes, and manipulate nodes and objects.

```
// Load the application descriptor from an external file
//Obtain GCMAplication
GCMApplication pad = PAGCMDeployment.loadApplicationDescriptor(new File(descriptor));

//Activate all Virtual Nodes
//Start the deployment of this application instance.
//Processes described in the GCM Application Descriptor are started on remote resources
//described by all GCM Deployment Descriptors XML files.
pad.startDeployment();

//Wait for all the virtual nodes to become ready
pad.waitReady();

//Obtain all the virtual nodes
 Map<String,GCMVirtualNode> vNodes = pad.getVirtualNodes();

//Get the first virtual node
 GCMVirtualNode vn =nodes.values().iterator().next();

// Get all nodes mapped to the target Virtual Node
List<Node> nodes =  vn.getCurrentNodes();

//Get the first node
Node node= nodes.get(0);
```

```
// Returns a reference to all AOs deployed on the target Node
Object[] node.getActiveObjects();

// Returns a reference to the ProActive Runtime (the JVM) where the
// node has been created
ProActiveRuntime part = node.getProActiveRuntime();

//Terminates all the ProActive Runtimes that have been started by this Application.
// Acquired resources are freed too.
pad.kill();
```

B.15. Exceptions

Functional exceptions with asynchrony

```
ProActive.tryWithCatch(MyException.class);
// Just before the try
try {
    // Asynchronous calls with exceptions
    .....
    // One can use PAException.throwArrivedException() and
    // PAException.waitForPotentialException() here

    // At the end of the try
    PAException.endTryWithCatch();
  } catch (MyException e) {
     // ...
  } finally {
     // At the beginning of the finally
     PAException.removeTryWithCatch();
}
```

B.16. Export Active Objects as Web services

ProActive allows active objects or components exportation as web services. The service is deployed on the local Jetty server or on another application server with a given URL. It is identified by its urn, an unique id of the service. It is also possible to choose the exported methods of the object.

The WSDL file matching the service will be accessible at http://localhost:8080/servlet/wsdl? for a service which name is 'a' and which id deployed on a web server which location is http://localhost:8080.

```
HelloWorld hw = (HelloWorld) PAActiveObject
      .newActive(HelloWorld.class.getName(), new Object[] {});

// As only one framework is supported for the moment,
// you can use either one of the two following methods
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getDefaultWebServicesFactory();
wsf = AbstractWebServicesFactory.getWebServicesFactory("cxf");

// If you want to use the local Jetty server, you can use
// AbstractWebServicesFactory.getLocalUrl() to get its url with
// its port number (which is random except if you have set the
// proactive.http.port variable)
```

```
WebServices ws = wsf.getWebServices("http://localhost:8080/");

ws.exposeAsWebService(hw, "MyHelloWorldService", new String[] { "putTextToSayAndConfirm",
      "putTextToSay", "sayText" });
```

B.17. Deploying a fault-tolerant application

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. The programmer can select **at deployment time** the most adapted protocol regarding the application and the execution environment.

A Fault-tolerant deployment descriptor

```
<ProActiveDescriptor>
  .......
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="Workers" property="multiple"
        ftServiceId="appli" />
      <virtualNode name="Failed" property="multiple"
        ftServiceId="resources" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="Workers">
        <jvmSet>
          <vmName value="Jvm1" />
          <vmName value="Jvm2" />
          <vmName value="Jvm3" />
        </jvmSet>
      </map>
      <map virtualNode="Failed">
        <jvmSet>
          <vmName value="JvmS1" />
          <vmName value="JvmS2" />
        </jvmSet>
      </map>
    </mapping>
  .....
  </deployment>
  .....
  <services>
      <serviceDefinition id="appli">
        <faultTolerance>
          <!-- Protocol selection: cic or pml -->
          <protocol type="cic"></protocol>
          <!-- URL of the fault-tolerance server -->
          <globalServer url="rmi://eon11:1100/FTServer"></globalServer>
          <!--  Average time in seconds between two consecutive checkpoints for each object -->
          <ttc value="15"></ttc>
        </faultTolerance>
      </serviceDefinition>
```

```
        <serviceDefinition id="resources">
          <faultTolerance>
            <protocol type="cic"></protocol>
            <globalServer url="rmi://eon11:1100/FTServer"></globalServer>
            <!-- URL of the resource server; all the nodes mapped on this virtual
        node will be registred in as resource nodes for recovery -->
            <resourceServer url="rmi://eon11:1100/FTServer" />
            <!--  Average time in seconds between two consecutive checkpoints for each object -->
            <ttc value="15"></ttc>
          </faultTolerance>
        </serviceDefinition>
      </services>
   .....
</ProActiveDescriptor>
```

Starting the fault-tolerance server

The global fault-tolerance server can be launched using the bin/startGlobalFTServer.[sh|bat] script, with 5 optional parameters:

- the protocol: -proto [cic|pml]. Default value is cic.
- the server name: -name [serverName]. Default name is FTServer.
- the port number: -port [portNumber]. Default port number is 1100.
- the fault detection period: -fdperiod [periodInSec], the time between two consecutive fault detection scanning. Default value is 10 sec.

B.18. Branch and Bound API

Firstly, create your own task:

```
import org.objectweb.proactive.extra.branchnbound.core.Task;
public class YourTask extends Task {

 public Result execute() {
   // Your code here for computing a solution
 }

 public Vector<Task> split() {
  // Your code for generating sub-tasks
  }

 public Result gather(Result[] results) {
  // Override optional
  // Default behavior based on the smallest gave by the compareTo
  }

 public void initLowerBound() {
  // Your code here for computing a lower bound
  }

 public void initUpperBound() {
  // Your code here for computing a lower bound
  }

 public int compareTo(Object arg) {
```

```
    // Strongly recommended to override this method
    // with your behavior
    }

}
```

How to interact with the framework from inside a task:

- Some class variables:

```
// to store your lower bound
protected Result initLowerBound;

// to store you upper bound
protected Result initUpperBound;

// set by the framework with the best current solution
protected Object bestKnownSolution;

// to interact with the framework (see below)
protected Worker worker;
```

- Interact with the framework (inside a Task):

```
// the worker will broadcast the solution in all Tasks
this.worker.setBestCurrentResult(newBestSolution);

// send a set of sub-tasks for computation to the framework
this.worker.sendSubTasksToTheManager(subTaskList);

// for a smart split, check for free workers
BooleanWrapper workersAvailable = this.worker.isHungry();
```

Secondly, choose your task queue:

- BasicQueueImpl: execute task in FIFO order.
- LargerQueueIml: execute task in larger order.
- Extend TaskQueue: your own one.

Finally, start the computation:

```
Task task = new YourTask(someArguments);
Manager manager = ProActiveBranchNBound.newBnB(task, nodes, LargerQueueImpl.class.getName());

// this call is asynchronous
Result futureResult = manager.start();
```

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

1. rootTask.initLowerBound(); // compute a first lower bound
2. rootTask.initUpperBound(); // compute a first upper bound
3. Task 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

B.19. File Transfer Deployment

File Transfer Deployment is a tool for transferring files at deployment time. This files are specified using the ProActive XML Deployment Descriptor in the following way:

```
<VirtualNode name='exampleVNode'FileTransferDeploy='example'/>
....
</deployment>
<FileTransferDefinitions>
  <FileTransfer id='example'>
     <file src='hello.dat' dest='world.dat'/>
     <dir src='exampledir' dest='exampledir'/>
  </FileTransfer>
  ...
</FileTransferDefinitions>
<infrastructure>
....
<processDefinition id='xyz'>
  <sshProcess>...
    <FileTransferDeploy='<emphasis
        role="bold">implicit'>
<!-- referenceID or keyword 'implicit' (inherit)-->
     <copyProtocol>processDefault, scp, rcp</<emphasis
        role="bold">copyProtocol>
     <sourceInfo prefix='/home/user'/>
     <destinationInfo prefix='/tmp' hostname='foo.org' username='smith' />
    </FileTransferDeploy>
  </sshProcess>
</processDefinition>
```

Bibliography

[ACC05] Isabelle Attali, Denis Caromel, and Arnaud Contes. *Deployment-based security for grid applications*. The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25. . LNCS. 2005. Springer Verlag.

[BBC02] Laurent Baduel, Francoise Baude, and Denis Caromel. *Efficient, Flexible, and Typed Group Communications in Java*. 28--36. Joint ACM Java Grande - ISCOPE 2002 Conference. Seattle. . 2002. ACM Press. ISBN 1-58113-559-8.

[BBC05] Laurent Baduel, Francoise Baude, and Denis Caromel. *Object-Oriented SPMD*. Proceedings of Cluster Computing and Grid. Cardiff, United Kingdom. . May 2005.

[BCDH05] Francoise Baude, Denis Caromel, Christian Delbe, and Ludovic Henrio. *A hybrid message logging-cic protocol for constrained checkpointability*. 644--653. Proceedings of EuroPar2005. Lisbon, Portugal. . LNCS. August-September 2005. Springer Verlag.

[BCHV00] Francoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. *Communicating mobile active objects in java*. 633--643. http://www-sop.inria.fr/oasis/Julien.Vayssiere/publications/18230633.pdf. Proceedings of HPCN Europe 2000. . LNCS 1823. May 2000. Springer Verlag.

[BCM+02] Francoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssiere. *Interactive and descriptor-based deployment of object-oriented grid applications*. 93--102. http://www-sop.inria.fr/oasis/personnel/Julien.Vayssiere/publications/hpdc2002_vayssiere.pdf. Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. Edinburgh, Scotland. . July 2002. IEEE Computer Society.

[BCM03] Francoise Baude, Denis Caromel, and Matthieu Morel. *From distributed objects to hierarchical grid components*. http://proactive.activeeon.com//userfiles/file/papers/HierarchicalGridComponents.pdf. International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November. Springer Verlag. . 2003. Lecture Notes in Computer Science, LNCS. ISBN ??.

[Car93] Denis Caromel. *Toward a method of object-oriented concurrent programming*. 90--102. http://citeseer.ist.psu.edu/300829.html. *Communications of the ACM*. 36. 9. 1993.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer Verlag. 2005.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Serpette. *Asynchronous and deterministic objects*. 123--134. http://doi.acm.org/10.1145/964001.964012. Proceedings of the 31st ACM Symposium on Principles of Programming Languages. . 2004. ACM Press.

[CKV98a] Denis Caromel, W. Klauser, and Julien Vayssiere. *Towards seamless computing and metacomputing in java*. 1043--1061. http://proactive.inria.fr/doc/javallCPE.ps. Concurrency Practice and Experience. . Geoffrey C. Fox. 10, (11--13). September-November 1998. Wiley and Sons, Ltd..

[HCB04] Fabrice Huet, Denis Caromel, and Henri E. Bal. *A High Performance Java Middleware with a Real Application*. http://proactive.inria.fr/doc/sc2004.pdf. Proceedings of the Supercomputing conference. Pittsburgh, Pensylvania, USA. . November 2004.

[BCDH04] F. Baude, D. Caromel, C. Delbe, and L. Henrio. *A fault tolerance protocol for asp calculus : Design and proof*. http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5246.pdf. Technical ReportRR-5246. INRIA. 2004.

[FKTT98] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. *A security architecture for computational grids*. 83--92. http://citeseer.ist.psu.edu/foster98security.html. ACM Conference on Computer and Communications Security. . 1998.

[CDD06c] Denis Caromel, Christian Delbe, and Alexandre di Costanzo. *Peer-to-Peer and Fault-Tolerance: Towards Deployment Based Technical Services* . Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture . Paris, France. . January 2006.

[CCDMCompFrame06] Denis Caromel, Alexandre di Costanzo, Christian Delbe, and Matthieu Morel. *Dynamically-Fulfilled Application Constraints through Technical Services - Towards Flexible Component Deployments* . Proceedings of HPC-GECO/CompFrame 2006, HPC Grid programming Environments and COmponents - Component and Framework Technology in High-Performance and Scientific Computing . Paris, France. . June 2006. IEEE.

[CCMPARCO07] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. *Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines*. *Parallel Computing Journal on Large Scale Grid*. 2007.

[PhD-Morel] Matthieu Morel. *Components for Grid Computing* . http://www-sop.inria.fr/oasis/personnel/Matthieu.Morel/publis/phd_thesis_matthieu_morel.pdf. PhD thesis. University of Nice Sophia-Antipolis. 2006.

[FACS-06] I. \vCern\'a, P. Va\vrekov\'a, and B. Zimmerova. "Component Substitutability via Equivalencies of Component-Interaction Automata". To appear in ENTCS. 2006.

[JavaA05] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. "A Component Model for Architectural Programming". 2005.

[Fractal04] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. "An Open Component Model and Its Support in Java.". 2004.

[ifip05] Alessandro Coglio and Cordell Green. "A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Appplets". 2005.

[STSLib07] Fabricio Fernandes and Jean-Claude Royer. "The STSLIB Project: Towards a Formal Component Model Based on STS". To appear in ENTCS. 2007.

[InterfaceAutomata2001] Luca de Alfaro, Tom Henzinger. "Interface automata". 2001.

[CCM] OMG. "CORBA components, version 3". 2002.

[Plasil02] F. Plasil and S. Visnovsky. "Behavior Protocols for Software Components". *IEEE Transactions on Software Engineering*. 28. 2002.

[Reussnerm] Reussner, Ralf H.. "Enhanced Component Interfaces to Support Dynamic Adaption and Extension". IEEE. 2001.

[JKP05] P. Jezek, J. Kofron, F. Plasil. "Model Checking of Component Behavior Specification: A Real Life Experience". Electronic Notes in Theoretical Computer Science (ENTCS). 2005.

[MB01] V. Mencl and T. Bures. "Microcomponent-based component controllers: A foundation for component aspects". APSEC. . Dec. 2005. IEEE Computer Society.

[SPC01] L. Seinturier, N. Pessemier, and T. Coupaye. "AOKell: an Aspect-Oriented Implementation of the Fractal Specifications". 2005.

Index

W