

ProActive

Programming, Composing, Deploying on the Grid



version 2.1 (October 2004)

GUIDED TOUR

This tour is a practical introduction to ProActive.

First you will get some practical experience on how to program using ProActive. This will help your understanding of the library, and of the concepts driving it.

Second, you will be guided through some examples to run on your computer; this way, you will get an illustrated introduction to some of the functionalities and facilities offered by the library.

Alternatively, you can just have a look at the second part, where you just have to run the applications. In that case, do not worry about answering the questions involving code modifications.

If you need further details on how the examples work, check the ProActive [applications](#) page.

Table of Contents

0. Installation and setup	1
ProActive Installation	2
Quick Start	2
To Test ProActive with the examples	2
To develop with ProActive	2
Download and expand the archive	2
Run a few examples for testing	3
Local Example 1: Hello world !	3
Local Example 2: Reader/Writer	3
Local Example 3: The Dining Philosophers	3
CLASSPATH to set when writing application using ProActive	3
Create a java.policy file to set permissions	4
Create a log4j configuration file	4
Troubleshooting and support	4
1. Hands-on programming	6
1.1. The client – server example	7
Hello world ! example	8
The two classes	8
The Hello class	8
The HelloClient Class	9
Hello World within the same VM	10
Hello World from another VM on the same host	10
Hello World from abroad: another VM on a different host	10
1.2. Initialization of the activity	12
Design of the application	12
Programming	12
InitializedHello	12
Execution	13
1.3. A simple migration example	14
1.3.1. Required conditions	14
1.3.2. design	14
1.3.3. Programming	15
a) the MigratableHello class	15
c) the client class	15
1.3.4. Execution	15
1.4. migration of graphical interfaces	17
Design of the application	17
Programming	17
HelloFrameController	18
HelloFrame	18
Execution	18
.....	19
2. Introduction to some of the functionalities of ProActive	19
2.1. Synchronization with ProActive	20
The readers-writers	20
1. start the application	20
2. look and check the effect of different policies : even, writer priority, reader priority	21
3. look at the code for programming such policies	21
4. Introduce a bug in the Writer Priority policy	22
The dining philosophers	22
1. start the philosophers application	22
2. understand the color codes	23
3. test the autopilot mode	24

Table of Contents

<u>2.1. Synchronization with ProActive</u>	
<u>4. test the manual mode</u>	24
<u>5. start the IC2D application</u>	24
<u>2.2. Parallel processing with ProActive</u>	28
<u>C3D : a parallel, distributed and collaborative 3D renderer</u>	28
1. start C3D	28
2. start a user	30
3. start a user from another machine	31
4. start IC2D to visualize the topology	32
5. drag-and-drop migration	33
6. start a new JVM in a computation	34
7. have a look at the source code for the main classes of this application :	34
<u>2.3. Migration of active objects</u>	35
<u>Mobile agents</u>	35
1. start the penguin application	35
2. start IC2D to see what is going on	35
3. add an agent	35
4. add several agents	35
5. move the control window to another user	36
<u>3.0. SPMD Programming</u>	37
<u>OO SPMD on a Jacobi example</u>	37
1. Execution and first glance at the Jacobi code	37
1.1 Source files: ProActive/src/org/objectweb/proactive/examples/jacobi	37
1.2 Execution	37
2. Modification and compilation	38
2.1 Source modification	38
2.2 Compilation	38
3. Detailed understanding of the OO SPMD Jacobi	38
3.1 Structure of the code	38
3.2 OO SPMD behavior	39
3.3 Adding a Method barrier for a step by step execution	39
3.4 Understanding various different kind of barriers	41
4. Virtual Nodes and Deployment descriptors	43
4.1 Virtual Nodes	43
4.2 XML Descriptors	43
4.3 Changing the descriptor	44
5. Execution on several machines and Clusters	44
5.1 Execution on several machines in the room	44
5.2 Execution on Clusters	44
<u>3. Conclusion</u>	45

0. Installation and setup

Follow the [link](#) for downloading and installing ProActive.

The programming exercises in the first part imply that you :

- Don't forget to add the required libraries to your classpath (i.e. the libraries contained in the ProActive/lib directory, as well as either the proactive.jar archive, or the compiled classes of proactive (better if you modify the source code)
- use a policy file, such as ProActive/scripts/unix/proactive.security.policy, with the JVM option `-Djava.security.policy=filelocation/proactive.java.policy`

Set the CLASSPATH as follow :

Under linux :

```
export
```

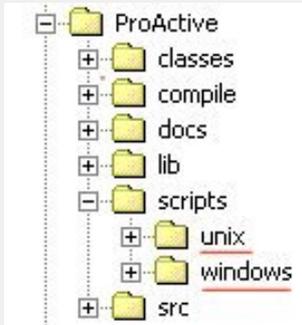
```
CLASSPATH=.:./ProActive_examples.jar:./ProActive.jar:./lib/bcel.jar:./lib/asm.jar:./lib/log4j.jar:./lib/xercesImpl.jar:./lib/fractal.jar:./lib/bouncycastle.jar
```

Under windows :

```
set CLASSPATH=.;.\ProActive_examples.jar;.\ProActive.jar;.\lib\bcel.jar;.\lib\asm.jar;.\lib\log4j.jar;.\lib\xercesImpl.jar;.\lib\fractal.jar;.\lib\bouncycastle.jar
```

Concerning [the second part](#) of the tutorial (examples of some functionalities):

- Note that the compilation is managed by [Ant](#) ; we suggest you use this tool to make modifications to the source code, while doing this tutorial. Nevertheless, you can just change the code and compile using `compile.sh <the example application>` (or `compile.bat` under windows)
- The examples used in the second part of this tutorial are provided in the /scripts directory of the distribution.
The scripts are platform dependant : `.sh` files on linux are equivalent to the `.bat` files on windows



ProActive Installation

ProActive is made available for [download](#) under a [LGPL license](#). ProActive requires the [JDK 1.4](#) or later to be installed on your computer. Please note that ProActive will NOT run with any version prior to 1.4 since some features introduced in JDK 1.4 are essential.

Quick Start

To Test ProActive with the examples

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

To develop with ProActive

- Download and unzip the ProActive archive
- Include in your CLASSPATH the following jar files ProActive/ProActive.jar, ProActive/lib/asm.jar, ProActive/lib/log4j.jar, ProActive/lib/xercesImpl.jar, ProActive/lib/fractal.jar, ProActive/lib/bouncycastle.jar
- Don't forget to launch the JVM with a [security policy file](#). You can also specify a [log4j configuration file](#) with the property `-Dlog4j.configuration=file:pathToFile`. If not specified a default logger that logs on the console will be created

Below are described the different steps in more details.

Download and expand the archive

You can download the archive file (a standard **zip** file) containing ProActive from the [download section](#) of the ProActive home page. You will be asked to accept the licence agreement and provide a few personal details including your email address. You will then within a few minutes receive an email.

Unzip the archive using your favorite ZIP program, such as [Winzip](#) under Windows or the [unzip](#) command-line utility on most Unix systems. Unzipping the archive creates a **ProActive** directory and all the files contained in the archive go into this directory and its subdirectories.

Here is a quick overview of the directory structure of the archive:

Directory or File	Description
ProActive.jar	ProActive bytecode that you need to include in the CLASSPATH in order to use ProActive
ProActive_examples.jar	The bytecode and resources of all examples included with ProActive. This jar file needs to be included in the CLASSPATH only when trying to run the examples. All examples rely on ProActive and therefore the ProActive.jar file must be included in the CLASSPATH as well. This is done automatically by the scripts driving the examples. The source code is also included in the src directory (see below)
ic2d.jar	The bytecode and resources of IC2D. This jar file needs to be included in the CLASSPATH only when trying to run the application IC2D. IC2D relies on ProActive and therefore the ProActive.jar file must be included in the CLASSPATH. This is done automatically by the scripts launching the application. The source code is also included in the src directory (see below)
lib	The external libraries used by ProActive
docs	ProActive documentation including the full api doc
scripts/unix	Unix <i>sh</i> scripts for running the examples
scripts/windows	Windows <i>.bat</i> batch files for running the examples
src	For source version only, the full source code of ProActive

compile

For source version only, the scripts to compile ProActive using Ant.

Run a few examples for testing

You can try to run the test applications provided with ProActive. Each example comes with a script to launch the application. Depending on you operating system, the script you need to launch is located either in `ProActive/scripts/unix` or `ProActive/scripts/windows`. The source code of all examples can be found in the directory `ProActive/src/org/objectweb/proactive/examples`.

Local Example 1: Hello world !

A simple example.

- script : `hello.sh` or `hello.bat`
- source : `ProActive/src/org/objectweb/proactive/examples/hello`

Local Example 2: Reader/Writer

This example is the ProActive version of the Readers/Writers canonical problem. To illustrate the ease-of-use of the ProActive model, different synchronization policies can be applied without even stopping the application. This example is based on a easy to use Swing GUI.

- script : `readers.sh` or `readers.bat`
- source : `ProActive/src/org/objectweb/proactive/examples/readers`

Local Example 3: The Dining Philosophers

This example is one possible implementation of the well-known *Dining Philosophers* synchronization problem. This example is based on a easy to use Swing GUI.

- script : `philosophers.sh` or `philosophers.bat`
- source : `ProActive/src/org/objectweb/proactive/examples/philosophers`

CLASSPATH to set when writing application using ProActive

Note that if you use the scripts provided with the distribution to run the examples you do not need to update your classpath.

In order to use **ProActive** in your application you need to place in your CLASSPATH the following jars files :

1. **ProActive.jar** : the library itself
2. **asm.jar** in `lib` directory : ASM the default bytecode manipulation framework used to dynamically generate the java bytecode of the Stubs
3. **log4j.jar** in `lib` directory : [Log4j](#) is the logging mechanism used in ProActive. You can define the way you want to log(console, file, GUI) by creating a `log4j` configuration file, like the one used in ProActive's example and located under `ProActive/scripts/unix` or `windows`. In that case, remind to run your java command with the property:
`-Dlog4j.configuration=file:pathToLog4jFile`
4. **xercesImpl.jar** in `lib` directory : Xerces is the library used to parse and validate xml files, like [Deployment Descriptors](#), [Configuration files](#) and [Component files](#)
5. **fractal.jar** in `lib` directory : Fractal is the component model used for [ProActive Components](#)
6. **bouncycastle.jar** in `lib` directory : This library is used by the [ProActive security framework](#)

You do not need to modify your CLASSPATH permanently as long as you include the two entries above using a Java IDE or a shell script.

In addition to the jar files above you may want to add the following jar files. None of them are used directly by the core fonctionnalities of ProActive but only in part of the library. Their are needed to compile all the code but they are not needed at runtime if those specific fonctionnalities are not used.

1. **bcel.jar** in `lib` directory : BCEL is also a bytecode manipulation framework that can be used optionnally (adding the `-DbyteCodeManipulator=BCEL` argument to the JVM)
2. **jini-core.jar**, **jini-ext.jar**, **reggie.jar** in `lib` directory: used to interface with Jini
3. **cog-jglobus.jar**, **cog-ogce.jar**, **cryptix.jar**, **cryptix32.jar**, **cryptix-asn1.jar**, **puretls.jar** in `lib` directory: used to interface with Globus

Create a java.policy file to set permissions

If you use the scripts provided with the distribution to run the examples an existing policy file named **proactive.java.policy** will be used by default

See [Permissions in the Java™ 2 SDK](#) to learn more about Java permissions. As a first approximation to run your code you can create a simple policy file granted all for everything :

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

Create a log4j configuration file

If you use the scripts provided with the distribution to run the examples an existing log4j file named **proactive-log4j** will be used by default

Example

```
# the default logging level is INFO

log4j.rootLogger=INFO, A1

#A1 uses PatternLayout
#and displays the associated message (%m)
#using the platform dependant separator (%n)
#Use %M for method names
#see log4j documentation for details

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%m %n

##### Change here default logging level on a
##### per-logger basis
##### usage is log4j.logger.className=Level, Appender
```

Troubleshooting and support

If you encounter any problem with installing ProActive and running the examples, please make sure you correctly followed all the steps described above. If it doesn't help, here is a list of the most common mistakes:

- **Permission denied when trying to launch scripts under Linux** Permissions do not allow to execute files. Just change the permissions with **chmod 755 *.sh**
- **Java complains about not being able to find ProActive's classes.** Your CLASSPATH environment variable does not contain the entry for the ProActive's or ASM's or Log4j's or Xerces' or Fractal's or BouncyCastle's classes. ProActive.jar, asm.jar, log4j.jar, xercesImpl.jar, fractal.jar, bouncycastle.jar must be in your CLASSPATH.
- **Java complains about denial of access.** If you get the following exceptions, you probably didn't change the file java.policy as described in step 4.

```
org.objectweb.proactive.NodeException:
java.security.AccessControlException: access denied
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
    at org.objectweb.proactive.core.node.rmi.RemoteNodeImpl.<init>(RmiNode.java:17)
    at org.objectweb.proactive.core.node.rmi.RemoteNodeFactory._createDefaultNode (RmiNodeFactory.java, Compiled Code)
    at org.objectweb.proactive.core.node.NodeFactory.createDefaultNode(NodeFactory.java:127)
    at org.objectweb.proactive.core.node.NodeFactory.getDefaultNode(NodeFactory.java:57)
    at org.objectweb.proactive.ProActive.newActive(ProActive.java:315)
    ...
```

```
Exception in thread "main" java.lang.ExceptionInInitializerError:
java.security.AccessControlException: access denied
(java.util.PropertyPermission user.home read)
    at java.security.AccessControlContext.checkPermission (AccessControlContext.java, Compiled Code)
    at java.security.AccessController.checkPermission(AccessController.java:403)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
    at java.lang.SecurityManager.checkPropertyAccess(SecurityManager.java:1243)
    at java.lang.System.getProperty(System.java:539)
    at org.objectweb.proactive.mop.MOPProperties.createDefaultProperties (MOPProperties.java:190)
    ...
```

- **Java complains log4j initialization** If you get the following message, you probably made a mistake when giving the -Dlog4j.configuration property to the java command. Be sure that the given path is right, try also to add file: before the path.

```
log4j:WARN No appender could be found for logger .....  
log4j:WARN Please initialize the log4j system properly
```

- **Examples and compilation do not work at all under Windows system:** Check if your java installation is not in a directory with spaces like C:\Program Files\java or C:\Documents and Settings\java. Indeed bat scripts do not run properly when JAVA_HOME is set to such directories. 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....

If you cannot solve the problem, feel free to email us for support at: proactive-support@sophia.inria.fr. Make sure you include a precise description of your problem along with a full copy of the error message you get.

1. Hands-on programming

Here is an introduction to programming with ProActive. It should give you a first flavor, and you will be able to see some more advanced examples in the section "introduction to some functionalities of the library".

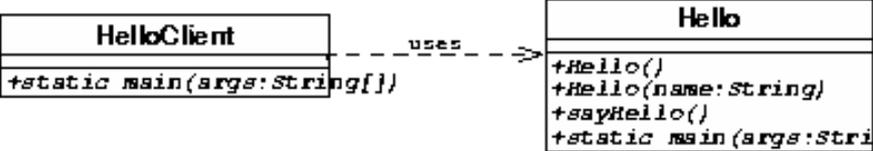
The program that we will develop is a kind of "helloworld" example. We will increase the complexity of the example, so that you familiarize yourself with different features of ProActive.

- First, we will code a ["client-server" application](#), the server being an active object.
- Second, we will see how we can [control the activity](#) of an active object.
- Third, we will [add mobility](#) to this active object.
- Eventually, we will [attach a graphical interface](#) to the active object, and we will show how to move the widget between virtual machines (like in the penguin example).

1.1. The client – server example

This example implements a very simple client-server application. You will find it [here](#), as it is the HelloWorld example earlier mentioned in this manual. A client object displays a `String` gotten from a remote server.

The corresponding class diagram is the following :



Hello world ! example

This example implements a very simple client–server application. A client object display a `String` received from a remote server. We will see how to write classes from which active and remote objects can be created, how to find a remote object and how to invoke methods on remote objects.

The two classes

Only two classes are needed: one for the server object `Hello` and one for the client that accesses it `HelloClient`.

The Hello class

This class implements server–side functionalities. Its creation involves the following steps:

- Provide an implementation for the required server–side functionalities
- Provide an empty, no–arg constructor
- Write a `main` method in order to instantiate one server object and register it with an URL.

Here is a possible implementation for the `Hello` class:

Hello.java

```
public class Hello {

    private String name;
    private String hi = "Hello world";
    private java.text.DateFormat dateFormat = new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

    public Hello() {
    }

    public Hello(String name) {
        this.name = name;
    }

    public String sayHello() {
        return hi + " at " + dateFormat.format(new java.util.Date())+
            " from node : " + org.objectweb.proactive.ProActive.getBodyOnThis().getNodeURL();
    }

    public static void main(String[] args) {
        // Registers it with an URL
        try {
            // Creates an active instance of class HelloServer on the local node
            Hello hello = (Hello)org.objectweb.proactive.ProActive.newActive(Hello.class.getName(), new Object[]{"remote"});
            java.net.InetAddress localhost = java.net.InetAddress.getLocalHost();
            org.objectweb.proactive.ProActive.register(hello, "/" + localhost.getHostName() + "/Hello");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Implement the required functionalities

Implementing any remotely–accessible functionality is simply done through normal Java methods in a normal Java class, in exactly the same manner it would have been done in a non–distributed version of the same class. This has to be contrasted with the RMI approach, where several more steps are needed:

- Define a remote interface for declaring the remotely–accessible methods.
- Rewrite the class so that it inherits from `java.rmi.server.UnicastRemoteObject`, which is the root class of all remote objects.
- Add remote exceptions handling to the code.

Why an empty no-arg constructor ?

You may have noticed that class `Hello` has a constructor with no parameters and an empty implementation. The presence of this empty no-arg constructor is imposed by ProActive and is actually a side-effect of ProActive's transparent implementation of active remote objects (as a matter of fact, this side-effect is caused by ProActive being implemented on top of a 100% Java metaobject protocol). If no such constructor is provided, active objects cannot be created. If no constructor at all is provided, active objects can still be created because, in this specific case, all Java compilers provide a default no-arg empty constructor. If a no-arg constructor is provided but its implementation is not empty, unwanted behavior may appear because the no-arg constructor is always called when an active object is created, whatever code the user can write.

Creating the remote `Hello` object

Now that we know how to write the class that implements the required server-side functionalities, let us see how to create the server object. In ProActive, there is actually no difference between a server and a client object as both are remote objects. Creating the active object is done through *instantiation-based creation*. We want this active object to be created on the current node, which is why we use `newActive` with only two parameters. In order for the client to obtain an initial reference onto this remote object, we need to register it in the registry (which is actually the well-known `rmiregistry`) with a valid RMI URL.

The `HelloClient` Class

The responsibility of this class is first to locate the remote server object, then to invoke a method on it in order to retrieve a message, and finally display that message.

HelloClient.java

```
public class HelloClient {

    public static void main(String[] args) {
        Hello myServer;
        String message;
        try {
            // checks for the server's URL
            if (args.length == 0) {
                // There is no url to the server, so create an active server within this VM
                myServer = (Hello)org.objectweb.proactive.ProActive.newActive(Hello.class.getName(), new Object[]{"local"});
            } else {
                // Lookups the server object
                System.out.println("Using server located on " + args[0]);
                myServer = (Hello)org.objectweb.proactive.ProActive.lookupActive(Hello.class.getName(), args[0]);
            }
            // Invokes a remote method on this object to get the message
            message = myServer.sayHello();
            // Prints out the message
            System.out.println("The message is : " + message);
        } catch (Exception e) {
            System.err.println("Could not reach/create server object");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Looking up a remote object

The operation of *lookup* simply means obtaining a reference onto an object from the URL it is bound to. The return type of method `Proactive.lookupActive()` is `Object`, then we need to cast it down into the type of the variable that holds the reference (`Hello` here). If no object is found at this URL, the call to `Proactive.lookupActive()` returns `null`.

Invoking a method on a remote object

This is exactly like invoking a method on a local object of the same type. The user does not have to deal with catching distribution-related exceptions like, for example, when using RMI or CORBA. Future versions of ProActive will provide an exception handler mechanism in order to process these exceptions in a separate place than the functional code. As class `String` is `final`, there cannot be any asynchronism here since the object returned from the call cannot be replaced by a future object (this restriction on `final` classes is imposed by ProActive's implementation).

Printing out the message

As already stated, the only modification brought to the code by ProActive is located at the place where active objects are created. All the rest of the code remains the same, which fosters software reuse.

Hello World within the same VM

In order to run both the client and server in the same VM, the client creates an active object in the same VM if it doesn't find the server's URL. The code snippet which instantiates the Server in the same VM is the following:

```
if (args.length == 0) {
    // There is no url to the server, so create an active server within this VM
    myServer = (Hello)org.objectweb.proactive.ProActive.newActive(Hello.class.getName(), new Object[]{"local"});
}
```

To launch the Client and the Server, just type:

```
linux> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j
org.objectweb.proactive.examples.hello.HelloClient
```

```
windows> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j
org.objectweb.proactive.examples.hello.HelloClient &
```

Hello World from another VM on the same host

Starting the server

Just start the main method in the Hello class.

```
linux> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j org.objectweb.proactive.examples.hello.Hello
```

```
windows> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j org.objectweb.proactive.examples.hello.Hello &
```

Launching the client

```
linux> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j
org.objectweb.proactive.examples.hello.HelloClient //localhost/Hello
```

```
windows> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j
org.objectweb.proactive.examples.hello.HelloClient //localhost/Hello
```

Hello World from abroad: another VM on a different host

Starting the server

Log on to the server's host, and launch the Hello class.

```
linux remoteHost> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j org.objectweb.proactive.examples.hello.Hello
```

```
windows remoteHost> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j org.objectweb.proactive.examples.hello.Hello &
```

Launching the client

Log on to the client Host, and launch the client

```
linux clientHost> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j
org.objectweb.proactive.examples.hello.HelloClient //remoteHost/Hello
```

```
windows clientHost> java -Djava.security.policy=scripts\unix\proactive.java.policy  
-Dlog4j.configuration=file:scripts\unix\proactive-log4j  
org.objectweb.proactive.examples.hello.HelloClient //remoteHost/Hello
```

1.2. Initialization of the activity

Active objects, as indicates their name, have an activity of their own (an internal thread).

It is possible to add pre and post processing to this activity, just by implementing the interfaces `InitActive` and `EndActive`, that define the methods `initActivity` and `endActivity`.

The following example will help you to understand how and when you can initialize and clean the activity.

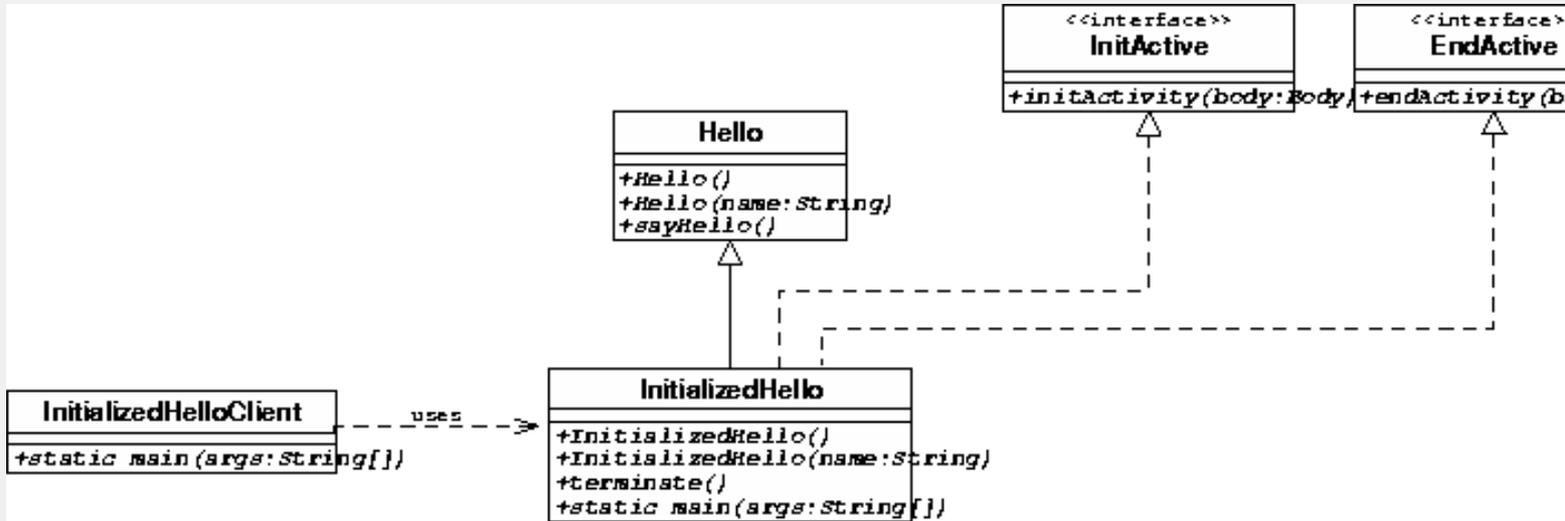
When instantiated, the activity of an object is automatically started, but it will first do what is written in the `initActivity` method.

Ending the activity can only be done from inside the active object (i.e. from a call to its own body). This is the reason why we have written a `terminate` method in the following example.

Design of the application

The `InitializedHello` class extends the `Hello` class, and implements the interfaces `InitActive` and `EndActive`. It acts as a server for the `InitializedHelloClient` class.

The main method is overridden so that it can instantiate the `InitializedHello` class



Programming

InitializedHello

The source code of the `InitializedHello` class is [here](#).

`initActivity` and `endActivity` here just log messages onto the console, so you can see when they are called.

`initActivity` is called at the creation of the active object, while `endActive` is called after the activity has terminated (thanks to the method `terminate`).

Here is the `initActivity` method :

```
public void initActivity(Body body) {
    System.out.println("I am about to start my activity");
}
```

Here is the `endActivity` method :

```
public void endActivity(Body body) {
    System.out.println("I have finished my activity");
}
```

The following code shows how to terminate the activity of the active object :

```
public void terminate() {
```

```

// the termination of the activity is done through a call on the
// terminate method of the body associated to the current active object
ProActive.getBodyOnThis().terminate();
}

```

The only differences from the the previous example is the classes instantiated, which are now `InitializedHello` (and not `Hello`) and `InitializedHelloClient`, and you will add a call to `hello.terminate()`.

InitializedHello: Code is [here](#).

InitializedHelloClient: Code is [here](#).

So, create `InitializedHelloClient.java` and `InitializedHello.java` in `src/org/objectweb/proactive/examples/hello`

Now compile all proactive sources

```

cd compile
windows>build.bat examples
linux>build examples
cd ..

```

Add `./classes` directory to `CLASSPATH` to use these two new source files

```

windows>set
CLASSPATH=.;.\classes;.\ProActive_examples.jar;.\ProActive.jar;.\lib\bcel.jar;.\lib\asm.jar;.\lib\log4j.jar;
linux>export
CLASSPATH=./classes:./ProActive_examples.jar:./ProActive.jar:./lib/bcel.jar:./lib/asm.jar:./lib/log4j.jar:

```

Execution

Execution is similar to the previous example; just use the `InitializedHelloClient` client class and `InitializedHello` server class.

Starting the server

```

linux> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j
org.objectweb.proactive.examples.hello.InitializedHello

```

```

windows> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j
org.objectweb.proactive.examples.hello.InitializedHello &

```

Launching the client

```

linux> java -Djava.security.policy=scripts/unix/proactive.java.policy
-Dlog4j.configuration=file:scripts/unix/proactive-log4j
org.objectweb.proactive.examples.hello.InitializedHelloClient //localhost/Hello

```

```

windows> java -Djava.security.policy=scripts\unix\proactive.java.policy
-Dlog4j.configuration=file:scripts\unix\proactive-log4j
org.objectweb.proactive.examples.hello.InitializedHelloClient //localhost/Hello

```

1.3. A simple migration example

This program is a very simple one : it creates an active object that migrates between virtual machines. It is a extension of the previous client–server example, the server now being mobile.

1.3.1. Required conditions

The conditions for MigratableHello to be a migratable active object are :

- it must have a constructor without parameters : this is a result of a ProActive restriction : the active object having to implement a no–arg constructor.
- implement the Serializable interface (as it will be transferred through the network).

Hello, the superclass, must be able to be serialized, in order to be transferred remotely. It does not have to implement directly java.io.Serializable, but its attributes should be serializable – or transient. For more information on this topic, check the [manual](#).

1.3.2. design

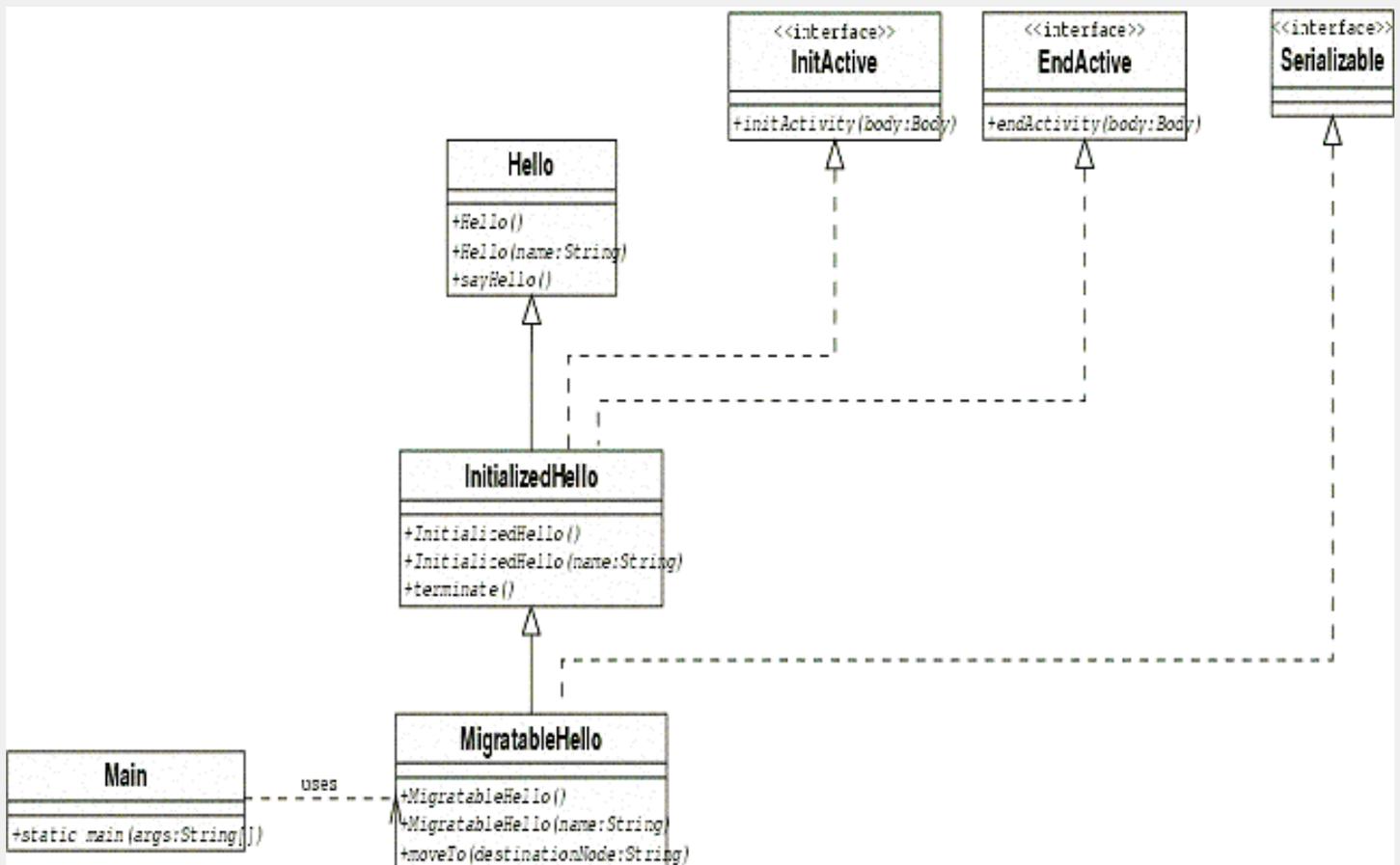
We want to further enhance InitializedHello it by making migratable : we'd like to be able to move it across virtual machines.

Thus, we create a MigratableHello class, that derives from InitializedHello. This class will implement all the non–functionnal behavior concerning the migration, for which this example is created. The Hello class (and InitializedHello) is left unmodified.

Note that the migration has to be initiated by the active object itself. This explains why we have to write the moveTo method in the code of MigratableHello – i.e. a method that contains an explicit call to the migration primitive. (cf [migration documentation](#))

MigratableHello also implements a factory method for instanciating itself as an active object : `static MigratableHello createMigratableHello(String : name)`

The class diagram for the application is the following :



1.3.3. Programming

a) the MigratableHello class

The code of the MigratableHello class is [here](#).

MigratableHello derives from the Hello class from the previous example

MigratableHello being the active object itself, it has to :

- implement the Serializable interface
- provide a no-arg constructor
- provide an implementation for using ProActive's migration mechanism.

A new method `getCurrentNodeLocation` is added for the object to tell the node where it resides..

A factory static method is added for ease of creation.

The migration is initiated by the `moveTo` method :

```
/** method for migrating
 * @param destination_node destination node
 */
public void moveTo(String destination_node) {
    System.out.println("\n-----");
    System.out.println("starting migration to node : " + destination_node);
    System.out.println("...");
    try {
        // THIS MUST BE THE LAST CALL OF THE METHOD
        ProActive.migrateTo(destination_node);
    } catch (MigrationException me) {
        System.out.println("migration failed : " + me.toString());
    }
}
```

Note that the call to the ProActive primitive `migrateTo` is the last one of the method `moveTo`. See the [manual](#) for more information.

c) the client class

The entry point of the program is written in a separate class : [MigratableHelloClient](#)

It takes as arguments the locations of the nodes the object will be migrated to.

The program calls the factory method of `MigratableHello` to create an instance of an active object. It then moves it from node to node, pausing for a while between the transfers.

1.3.4. Execution

- start several nodes using the `startnode` script.

```
windows>cd scripts/windows
                                startNode.bat //localhost/n1
                                startNode.bat //localhost/n2

linux>cd scripts/linux
    ./startNode.sh //localhost/n1
    ./startNode.sh //localhost/n2
```

- compile and run the program (run `MigratableHelloClient`), passing in parameter the urls of the nodes you'd like the agent to migrate to.

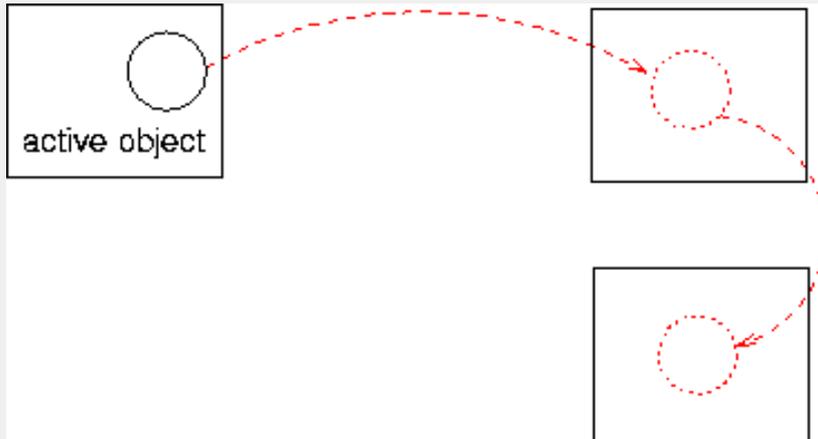
```
cd compile
windows>build.bat examples
linux>build examples
cd ..
```

```
linux>java -Djava.security.policy=scripts/unix/proactive.java.policy
```

```
-Dlog4j.configuration=file:scripts/unix/proactive-log4j  
org.objectweb.proactive.examples.hello.MigratableHelloClient //localhost/n1 //localhost/n2
```

```
windows>java -Djava.security.policy=scripts\unix\proactive.java.policy  
-Dlog4j.configuration=file:scripts\unix\proactive-log4j  
org.objectweb.proactive.examples.hello.MigratableHelloClient //localhost/n1 //localhost/n2
```

– observe the instance of MigratableHello migrating :



During the execution, a default node is first created. It then hosts the created active object. Then the active object is migrated from node to node, each time returning "hello" and telling the client program where it is located.

1.4. migration of graphical interfaces

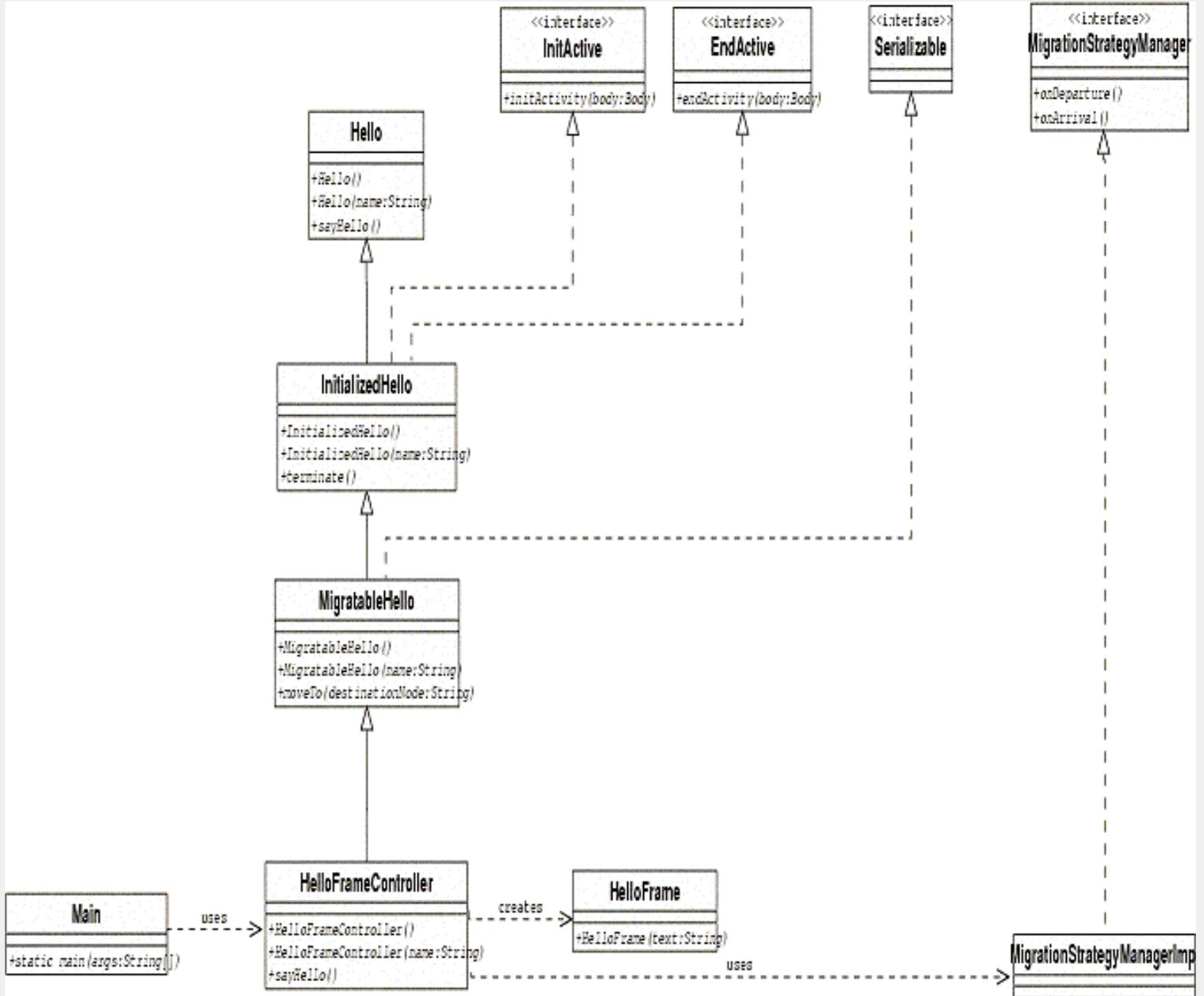
Graphical interfaces are not serializable, yet it is possible to migrate them with ProActive.

The idea is to associate the graphical object to an active object. The active object will control the activation and deactivation of this graphical entity during migrations.

Of course, this is a very basic example, but you can later build more sophisticated frames.

Design of the application

We will write a new active object class, that extends MigratableHello. The sayHello method will create a window containing the hello message. This window is defined in the class HelloFrame



Programming

HelloFrameController

The code of the HelloFrameController is [here](#).

This class extends MigratableHello, and adds an activity and a migration strategy manager to the object . It creates a graphical frame upon call of the sayHello method.

Here we have a more complex migration process than with the previous example. We need to make the graphical window disappear before and reappear in a new location after the migration (in this example though, we wait for a call to sayHello). The migration of the frame is actually controlled by a MigrationStrategyManager, that will be attached to the body of the active object.. An ideal location for this operation is the initActivity method (from InitActive interface), that we override :

```
/**
 * This method attaches a migration strategy manager to the current active object.
 * The migration strategy manager will help to define which actions to take before
 * and after migrating
 */
public void initActivity(Body body) {
    // add a migration strategy manager on the current active object
    migrationStrategyManager = new MigrationStrategyManagerImpl((Migratable) ProActive.getBodyOnThis());
    // specify what to do when the active object is about to migrate
    // the specified method is then invoked by reflection
    migrationStrategyManager.onDeparture("clean");
}
```

The MigrationStrategyManager defines methods such as "onDeparture", that can be configured in the application. For example here, the method "clean" will be called before the migration, conveniently killing the frame :

```
public void clean() {
    System.out.println("killing frame");
    helloFrame.dispose();
    helloFrame = null;
    System.out.println("frame is killed");
}
```

HelloFrame

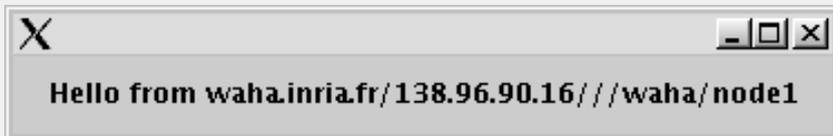
This is an example of a graphical class that could be associated with the active object. [Here](#) is the code.

Execution

- Similarly to the simple migration example (use the [HelloFrameController](#) class), you will start remote nodes and specify a migration path.
- you have 2 ways for handling the display of the graphical objects :

- ◆ look on the display screens of the machines
- ◆ export the displays : in startNode.sh, you should add the following lines before the java command :

```
DISPLAY=myhost:0
export DISPLAY
```



The displayed window : it just contains a text label with the location of the active object.

2. Introduction to some of the functionalities of ProActive

This chapter will present some of the facilities offered by ProActive, namely :

- [synchronization](#)
- [parallel processing](#)
- [migration](#)

2.1. Synchronization with ProActive

ProActive provides an advanced synchronization mechanism that allows an easy and safe implementation of potentially complex synchronization policies.

This is illustrated by two examples :

- The [readers and the writers](#)
- The [dining philosophers](#)

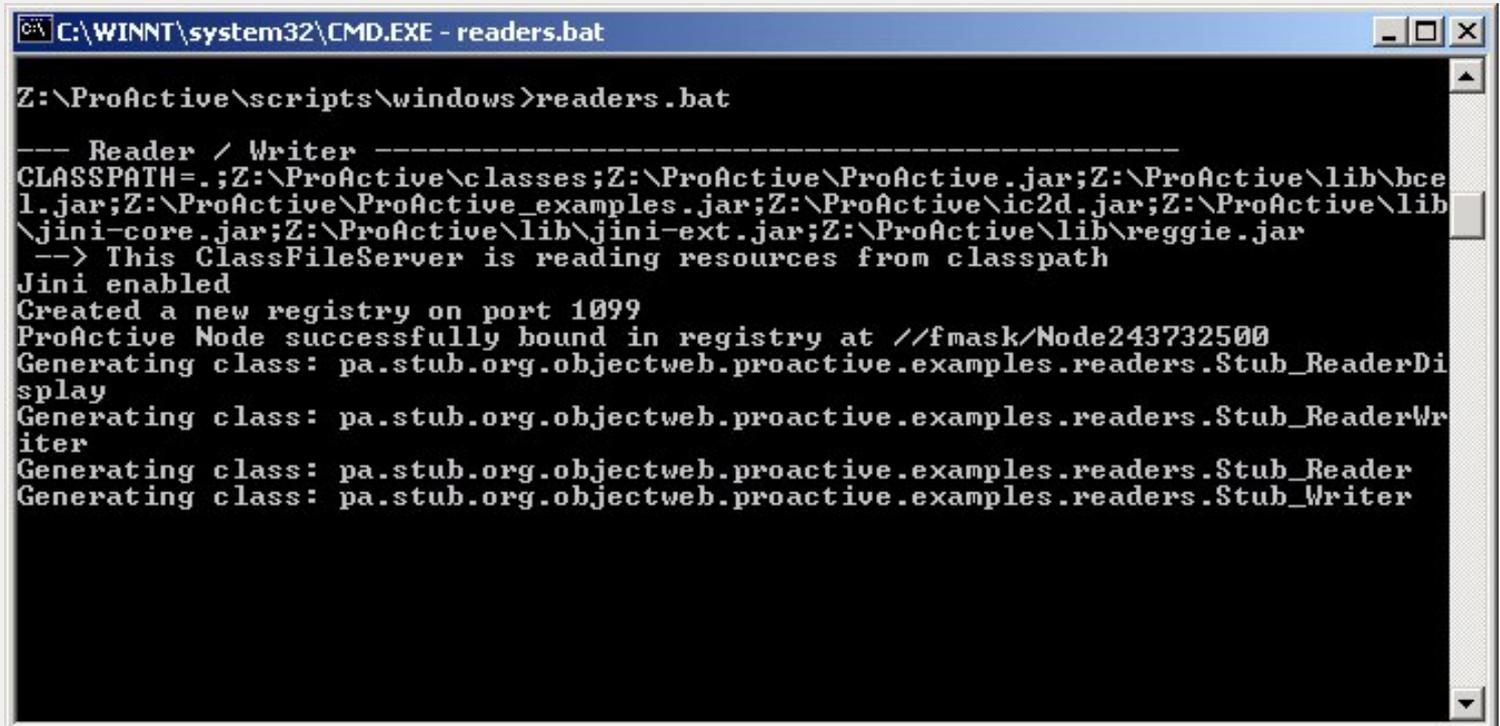
The readers–writers

The readers and the writers want to access the same data. In order to allow concurrency while ensuring the consistency of the readings, accesses to the data have to be synchronized upon a specified policy. Thanks to ProActive, the accesses are guaranteed to be allowed sequentially.

The [implementation with ProActive](#) uses 3 active objects : Reader, Writer, and the controller class (ReaderWriter).

1. start the application

using the readers script

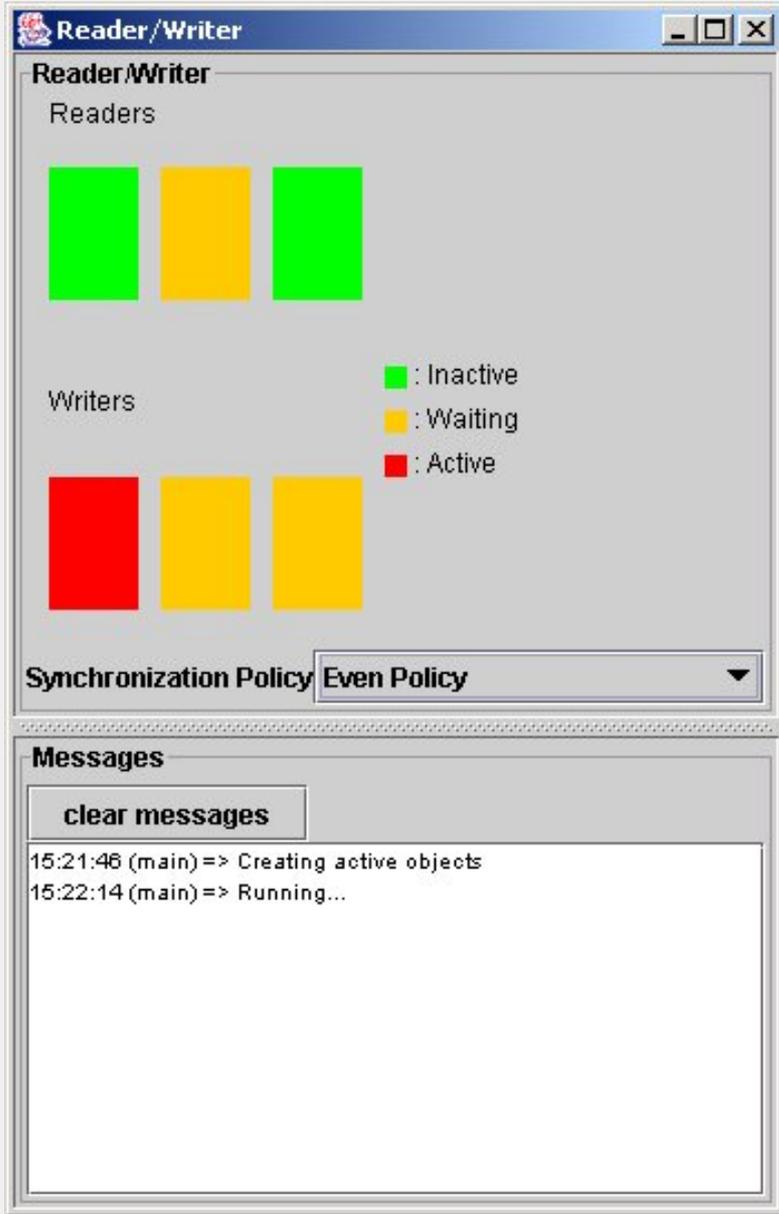


```
C:\WINNT\system32\CMD.EXE - readers.bat

Z:\ProActive\scripts\windows>readers.bat

--- Reader / Writer -----
CLASSPATH=.;Z:\ProActive\classes;Z:\ProActive\ProActive.jar;Z:\ProActive\lib\bce
l.jar;Z:\ProActive\ProActive_examples.jar;Z:\ProActive\ic2d.jar;Z:\ProActive\lib
\jini-core.jar;Z:\ProActive\lib\jini-ext.jar;Z:\ProActive\lib\reggie.jar
--> This ClassFileServer is reading resources from classpath
Jini enabled
Created a new registry on port 1099
ProActive Node successfully bound in registry at //fmask/Node243732500
Generating class: pa.stub.org.objectweb.proactive.examples.readers.Stub_ReaderDi
splay
Generating class: pa.stub.org.objectweb.proactive.examples.readers.Stub_ReaderWr
iter
Generating class: pa.stub.org.objectweb.proactive.examples.readers.Stub_Reader
Generating class: pa.stub.org.objectweb.proactive.examples.readers.Stub_Writer
```

ProActive starts a node (i.e. a JVM) on the current machine, and creates 3 Writer, 3 Reader, a ReaderWriter (the controller of the application) and a ReaderDisplay, that are active objects.



a GUI is started that illustrates the activities of the Reader and Writer objects.

2. look and check the effect of different policies : even, writer priority, reader priority

What happens when priority is set to "reader priority" ?

3. look at the code for programming such policies

in `org.objectweb.proactive.examples.readers.ReaderWriter.java`

More specifically, look at the routines in :

```
public void evenPolicy(org.objectweb.proactive.Service service)
```

```
public void readerPolicy(org.objectweb.proactive.Service service)
```

```
public void writerPolicy(org.objectweb.proactive.Service service)
```

Look at the inner class `MyRequestFilter` that implements `org.objectweb.proactive.core.body.request.RequestFilter`

How does it work?

4. Introduce a bug in the Writer Priority policy

For instance, let several writers go through at the same time.

- observe the Writer Policy policy before recompiling
- recompile (using compile.sh readers or compile.bat readers)
- observe that stub classes are regenerated and recompiled
- observe the difference due to the new synchronization policy : what happens now?
- correct the bug and recompile again ; check that everything is back to normal

The dining philosophers

The "dining philosophers" problem is a classical exercise in the teaching of concurrent programming. The goal is to avoid deadlocks.

We have provided [an illustration of the solution](#) using ProActive, where all the philosophers are active objects, as well as the table (controller) and the dinner frame (user interface).

1. start the philosophers application

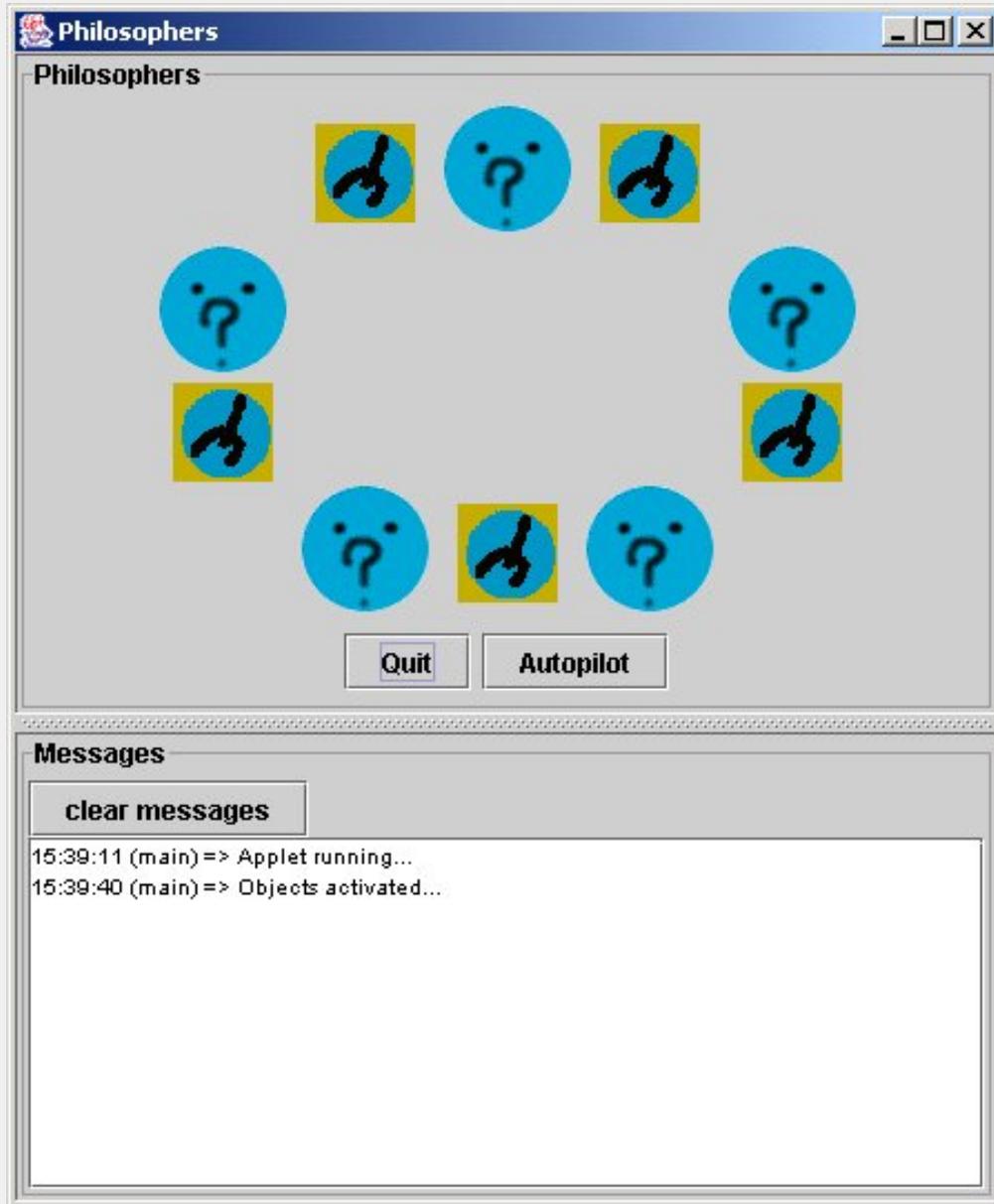
with philosophers.sh or philosophers.bat

```

C:\WINNT\system32\CMD.EXE - philosophers.bat
Z:\ProActive\scripts\windows>philosophers.bat

--- Philosophers -----
CLASSPATH=.;Z:\ProActive\classes;Z:\ProActive\ProActive.jar;Z:\ProActive\lib\bce
l.jar;Z:\ProActive\ProActive_examples.jar;Z:\ProActive\ic2d.jar;Z:\ProActive\lib
\jini-core.jar;Z:\ProActive\lib\jini-ext.jar;Z:\ProActive\lib\reggie.jar
--> This ClassFileServer is reading resources from classpath
Jini enabled
Created a new registry on port 1099
ProActive Node successfully bound in registry at //fmask/Node2034812486
Generating class: pa.stub.org.objectweb.proactive.examples.philosophers.Stub_Din
nerLayout
Generating class: pa.stub.org.objectweb.proactive.examples.philosophers.Stub_Tab
le
Generating class: pa.stub.org.objectweb.proactive.examples.philosophers.Stub_Phi
losopher
-
  
```

ProActive creates a new node and instantiates the active objects of the application : DinnerLayout, Table, and Philosopher



the GUI is started.

2. understand the color codes

		<i>philosophing</i>
<i>Philosophers</i>		<i>hungry, wants the fork !</i>
		<i>eating</i>
<i>Forks</i>		<i>taken</i>
		<i>free</i>

3. test the autopilot mode

The application runs by itself without encountering a deadlock.

4. test the manual mode

Click on the philosophers' heads to switch their modes

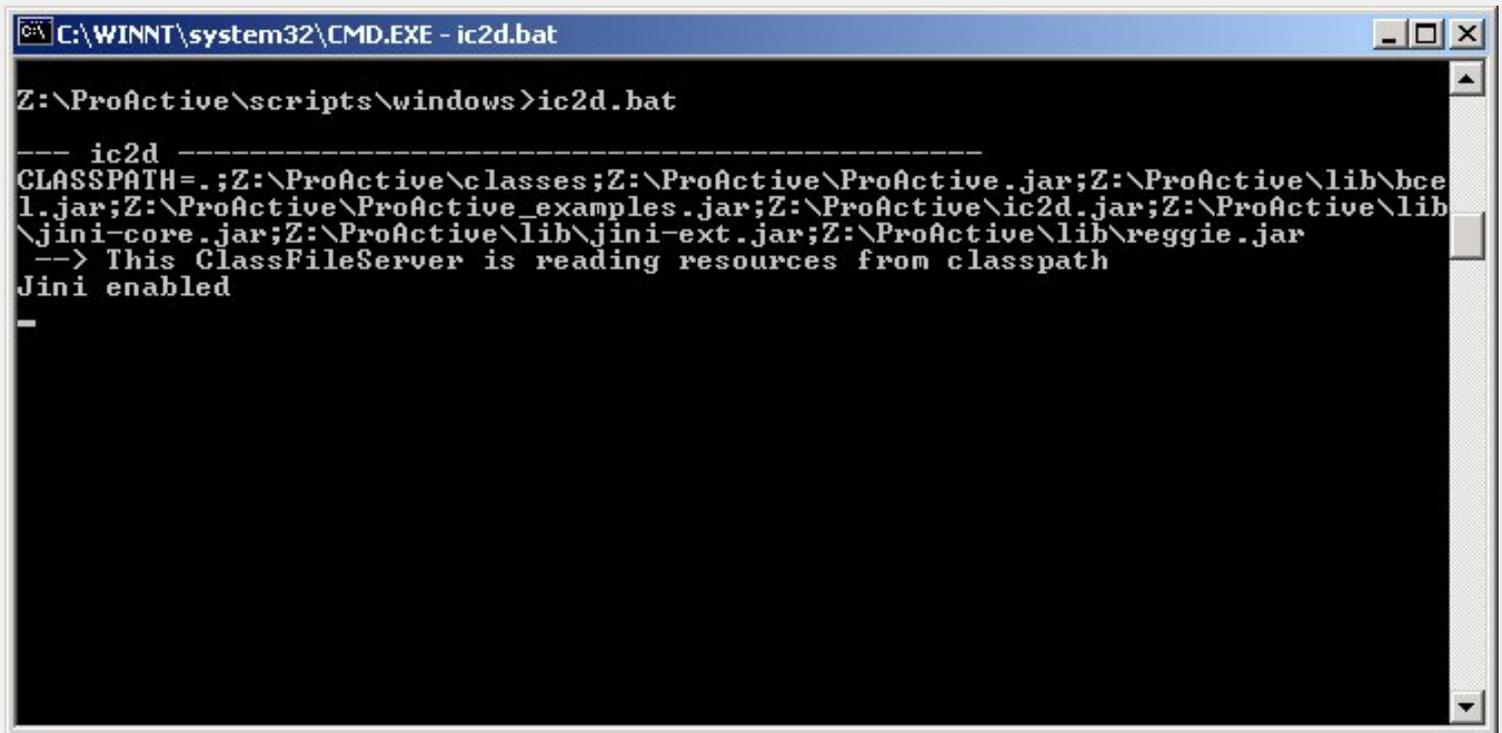
Test that there are no deadlocks!

Test that you can starve one of the philosophers (i.e. the others alternate eating and thinking while one never eats!)

5. start the IC2D application

[IC2D](#) is a graphical environment for monitoring and steering of distributed and metacomputing applications.

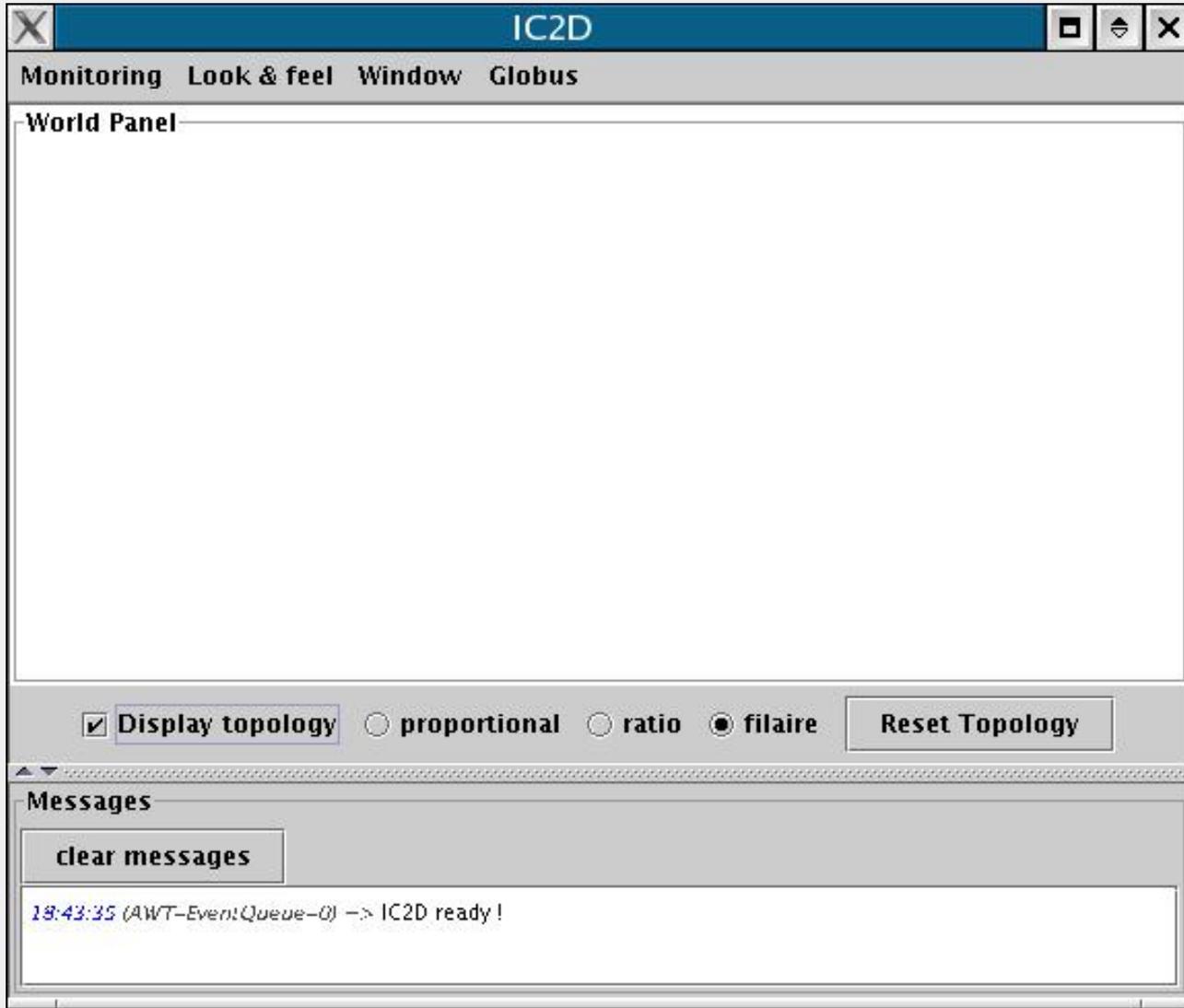
– being in the autopilot mode, start the IC2D visualization application (using ic2d.sh or ic2d.bat)



```
C:\WINNT\system32\CMD.EXE - ic2d.bat

Z:\ProActive\scripts\windows>ic2d.bat

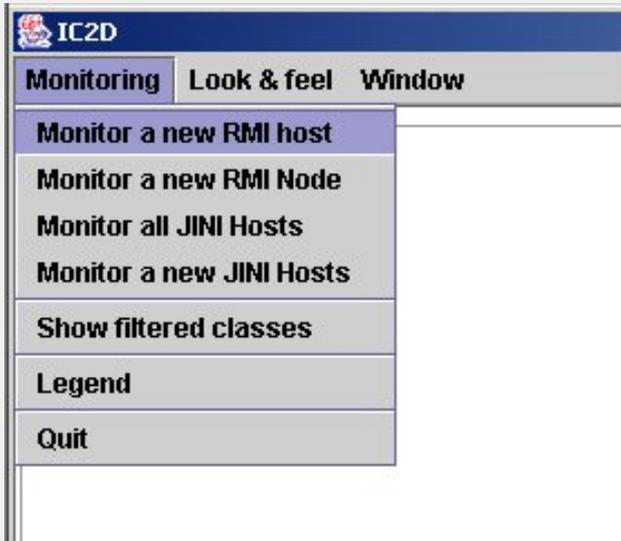
--- ic2d -----
CLASSPATH=.;Z:\ProActive\classes;Z:\ProActive\ProActive.jar;Z:\ProActive\lib\bce
l.jar;Z:\ProActive\ProActive_examples.jar;Z:\ProActive\ic2d.jar;Z:\ProActive\lib
\jini-core.jar;Z:\ProActive\lib\jini-ext.jar;Z:\ProActive\lib\reggie.jar
--> This ClassFileServer is reading resources from classpath
Jini enabled
_
```



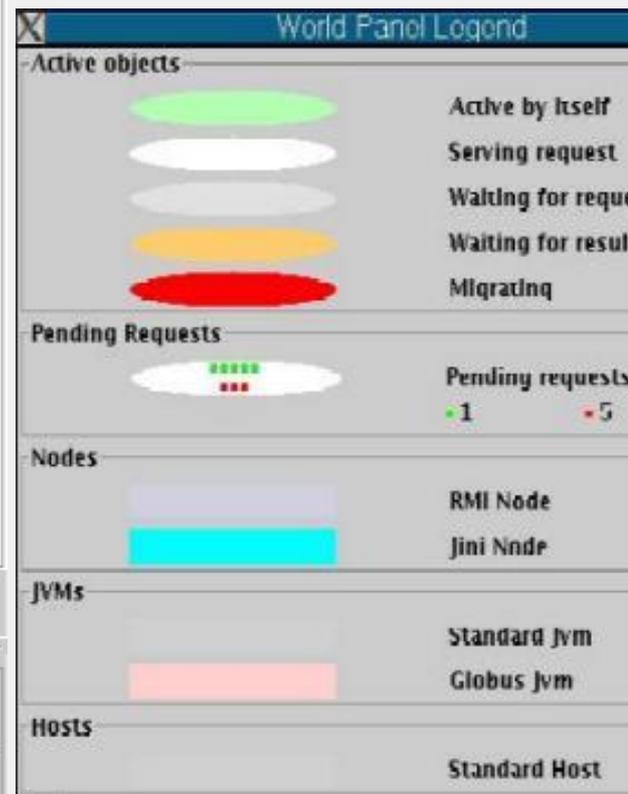
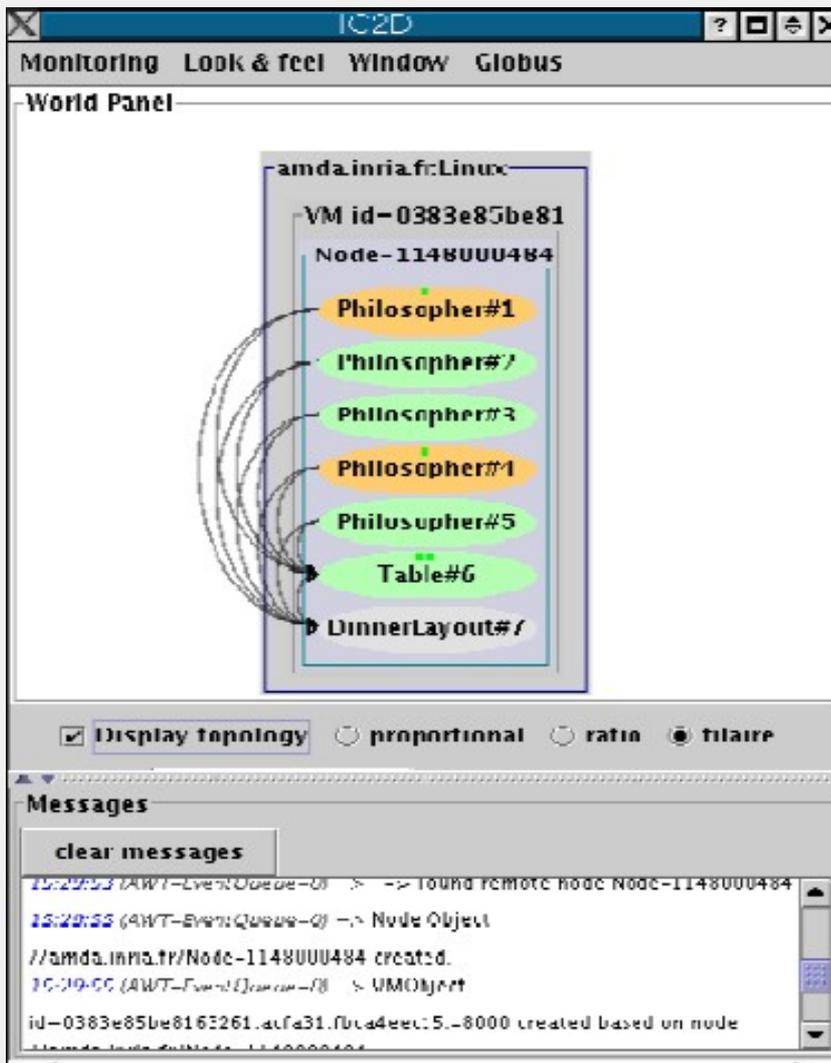
the ic2d GUI is started. It is composed of 2 panels : the main panel and the events list panel

– acquire you current machine

menu monitoring – monitor new RMI host



It is possible to visualize the status of each active object (processing, waiting etc...), the communications between active objects, and the topology of the system (here all active objects are in the same node) :



2.2. Parallel processing with ProActive

Distribution is often used for CPU-intensive applications, where parallelism is a key for performance.

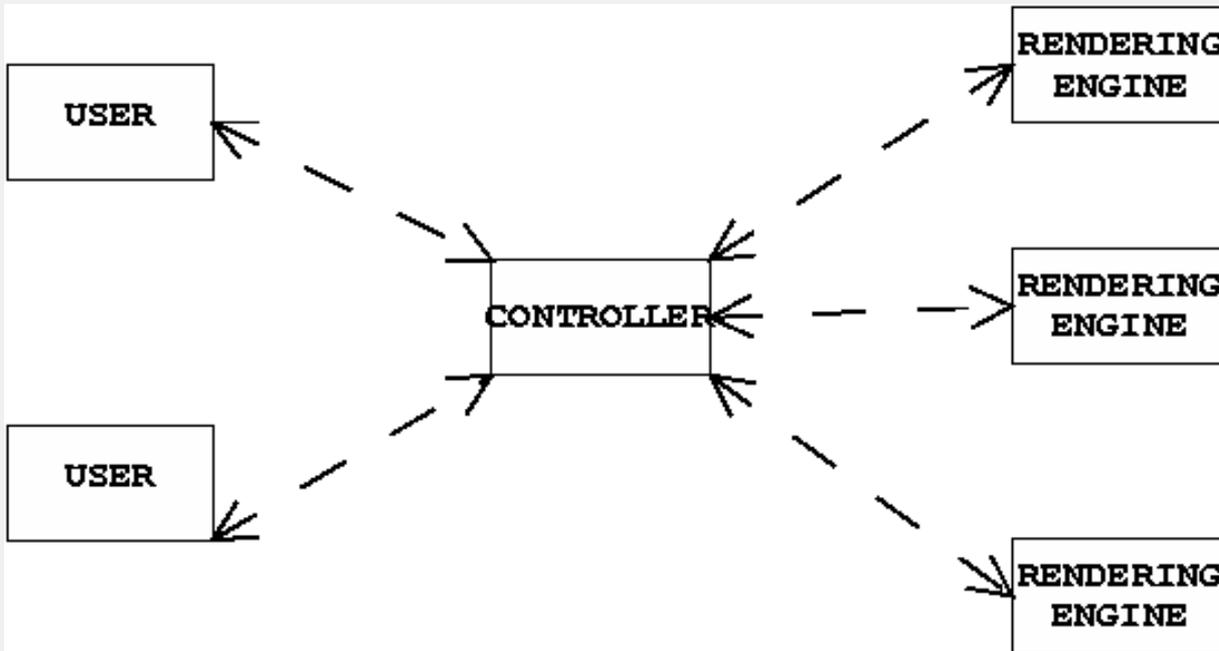
A typical application is [C3D](#).

Note that parallelisation of programs can be facilitated with ProActive, thanks to [asynchronism method calls](#), as well as [group communications](#).

C3D : a parallel, distributed and collaborative 3D renderer

[C3D](#) is a Java benchmark application that measures the performance of a 3D raytracer renderer distributed over several Java virtual machines using Java RMI. It showcases some of the benefits of ProActive, notably the ease of distributed programming, and the speedup through parallel calculation.

Several users can collaboratively view and manipulate a 3D scene. The image of the scene is calculated by a dynamic set of rendering engines using a raytracing algorithm, everything being controlled by a central dispatcher.

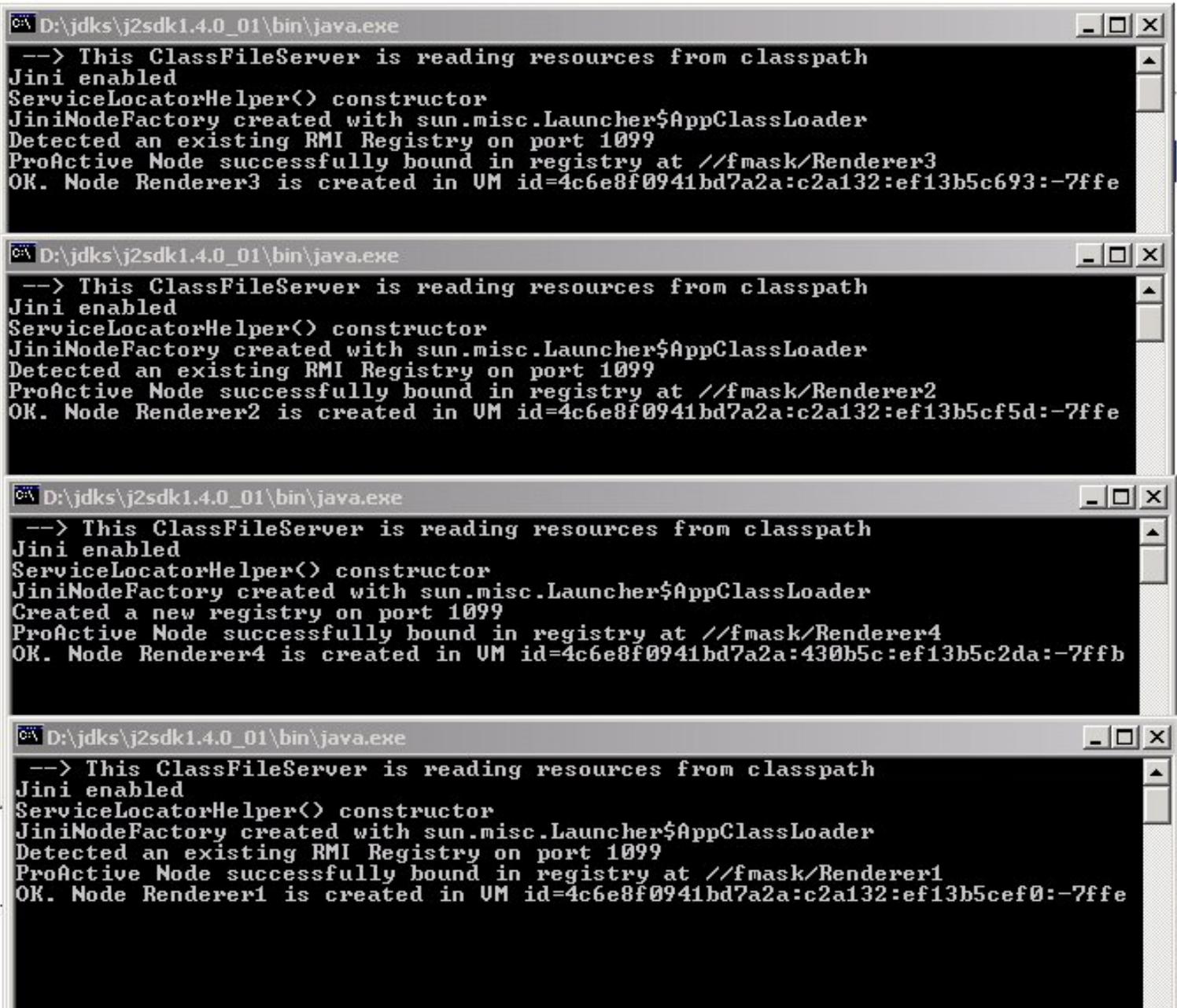


the active objects in the c3d application

1. start C3D

using the script `c3d_no_user`

A "Dispatcher" object is launched (ie a centralized server) as well as 4 "Renderer" objects, that are active objects to be used for parallel rendering.



The image displays four terminal windows, each showing the execution of a Java application. Each window has a title bar with the path 'D:\jdk\j2sdk1.4.0_01\bin\java.exe'. The output of each window is as follows:

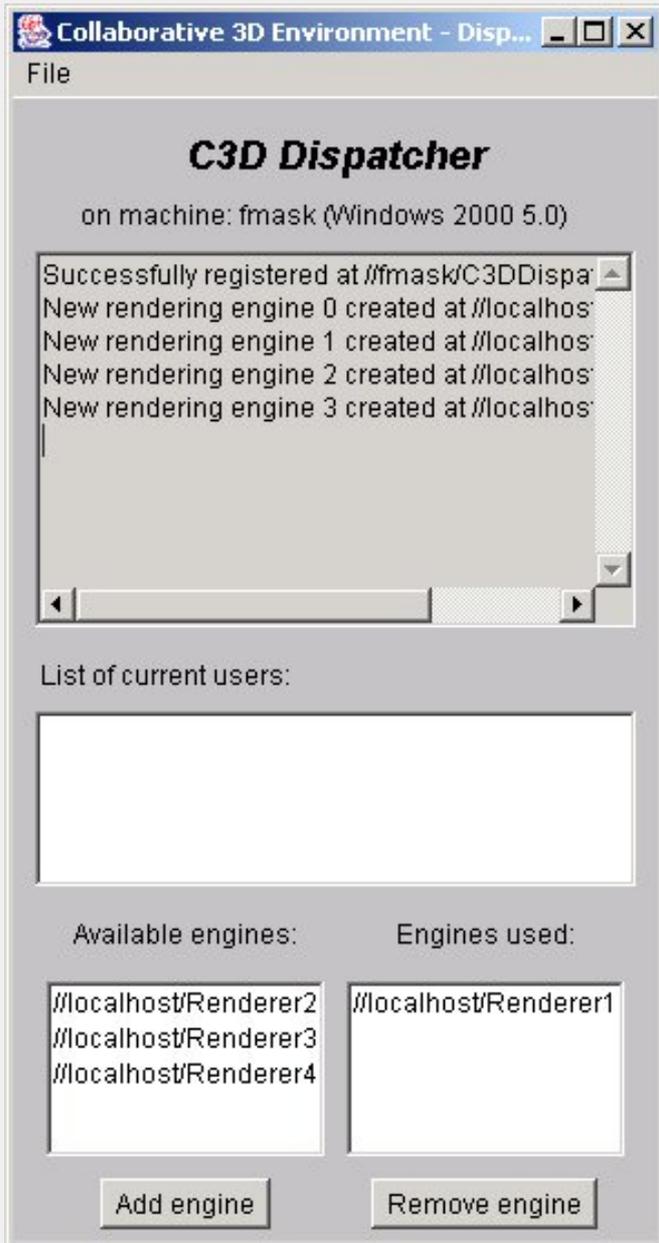
```
D:\jdk\j2sdk1.4.0_01\bin\java.exe
--> This ClassFileServer is reading resources from classpath
Jini enabled
ServiceLocatorHelper() constructor
JiniNodeFactory created with sun.misc.Launcher$AppClassLoader
Detected an existing RMI Registry on port 1099
ProActive Node successfully bound in registry at //fmask/Renderer3
OK. Node Renderer3 is created in UM id=4c6e8f0941bd7a2a:c2a132:ef13b5c693:-7ffe
```

```
D:\jdk\j2sdk1.4.0_01\bin\java.exe
--> This ClassFileServer is reading resources from classpath
Jini enabled
ServiceLocatorHelper() constructor
JiniNodeFactory created with sun.misc.Launcher$AppClassLoader
Detected an existing RMI Registry on port 1099
ProActive Node successfully bound in registry at //fmask/Renderer2
OK. Node Renderer2 is created in UM id=4c6e8f0941bd7a2a:c2a132:ef13b5cf5d:-7ffe
```

```
D:\jdk\j2sdk1.4.0_01\bin\java.exe
--> This ClassFileServer is reading resources from classpath
Jini enabled
ServiceLocatorHelper() constructor
JiniNodeFactory created with sun.misc.Launcher$AppClassLoader
Created a new registry on port 1099
ProActive Node successfully bound in registry at //fmask/Renderer4
OK. Node Renderer4 is created in UM id=4c6e8f0941bd7a2a:430b5c:ef13b5c2da:-7ffb
```

```
D:\jdk\j2sdk1.4.0_01\bin\java.exe
--> This ClassFileServer is reading resources from classpath
Jini enabled
ServiceLocatorHelper() constructor
JiniNodeFactory created with sun.misc.Launcher$AppClassLoader
Detected an existing RMI Registry on port 1099
ProActive Node successfully bound in registry at //fmask/Renderer1
OK. Node Renderer1 is created in UM id=4c6e8f0941bd7a2a:c2a132:ef13b5cef0:-7ffe
```

the 4 renderers are launched



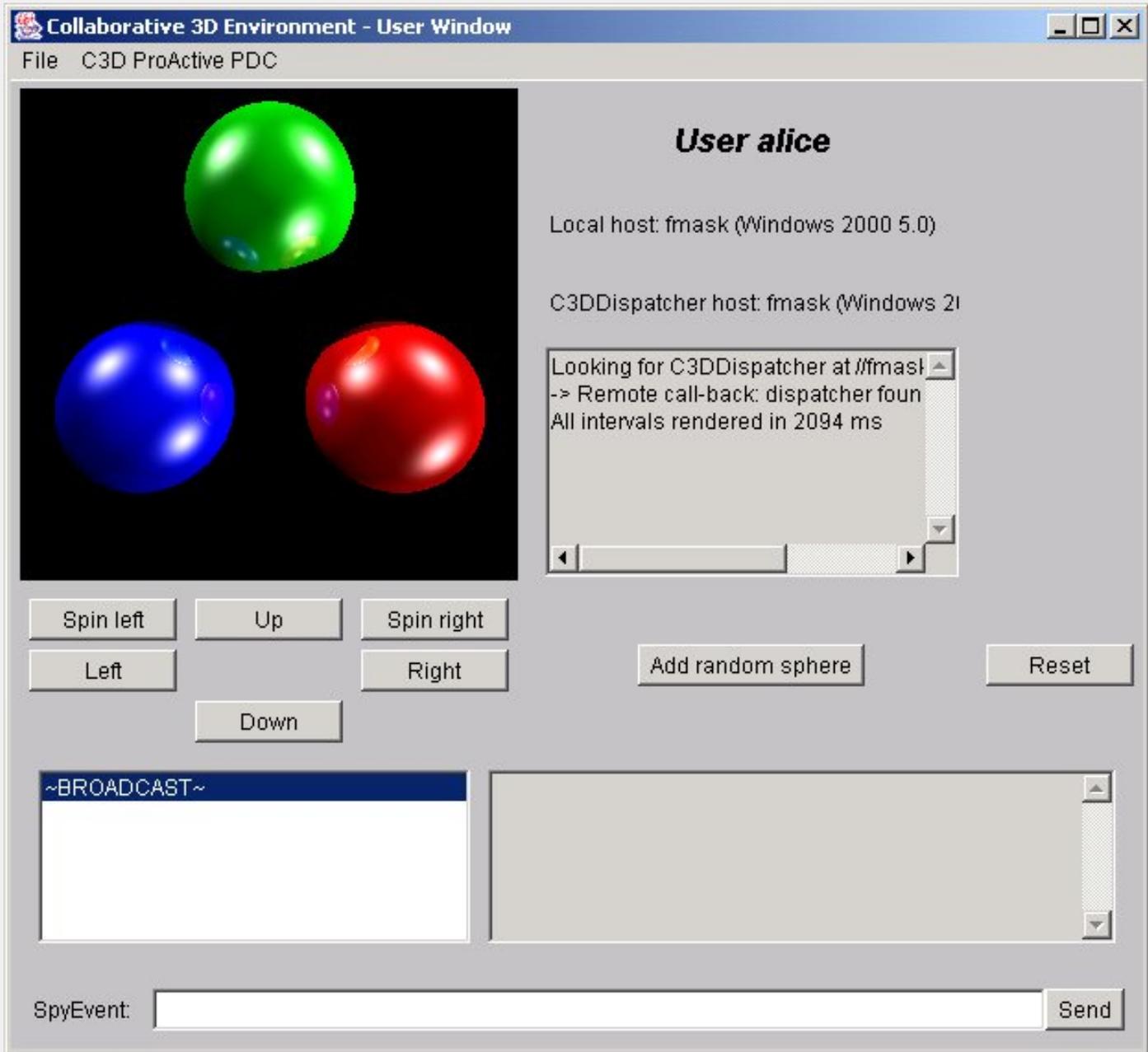
the dispatcher GUI is launched

The bottom part of the window allows the addition and removal of renderers.

2. start a user

using `c3d_add_user`

– connect on the current host (proposed by default) by just giving your name



for example the user "alice"

- spin the scene, add a random sphere, and observe how the action takes place immediately
- add and remove renderers, and observe the effect on the "speed up" indication from the user window.

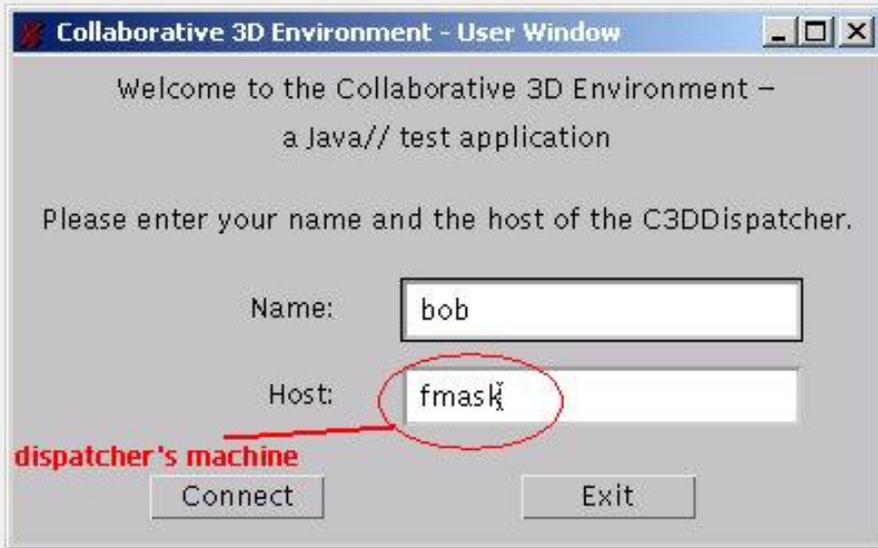
Which configuration is the fastest for the rendering?

Are you on a multi-processor machine?

** you might not perceive the difference of the performance. The difference is better seen with more distributed nodes and objects (for example on a cluster with 30+ renderers).*

3. start a user from another machine

using the `c3d_add_user` script, and specifying the host (NOT set by default)



If you use rlogin, make sure the DISPLAY is properly set.

You must use the same version of ProActive on both machines!

– test the collaborative behavior of the application when several users are connected.

Notice that a collaborative consensus must be reached before starting some actions (or that a timeout occurred).

4. start IC2D to visualize the topology

– to visualize all Active objects, you need to acquire ("monitoring" menu) :

- the machine on which you started the "Dispatcher"
- the machine on which you started the second user



- add random spheres for instance, and observe messages (Requests) between Active Objects.
- add and remove renderers, and check graphically whether the corresponding Active Objects are contacted or not, in order to achieve the rendering.
- you can textually visualize this information by activating "add event timeline for this WorldObject" on the World panel with the right mouse button, and then "show the event list window" on the top menu window

5. drag-and-drop migration

- from IC2D, you can drag-and-drop active objects from one JVM to another. Click the right button on a C3DRenderingEngine, and drag and drop it in another JVM. Observe the migration taking place.
- add a new sphere, using all rendering engines, and check that the messages are still sent to the active object that was asked to migrate.

– as migration and communications are implemented in a fully compatible manner, you can even migrate with IC2D an active object while it is communicating (for instance when a rendering action is in progress). Give it a try!

Since version 1.0.1 of the C3D example, you can also migrate the client windows!

6. start a new JVM in a computation

manually you can start a new JVM – a "Node" in the ProActive terminology – that will be used in a running system.

– on a different machine, or by remote login on another host, start another Node, named for instance NodeZ :

```
under linux : startNode.sh rmi://mymachine/NodeZ & (or startNode.bat
rmi://mymachine/NodeZ)
```

The node should appear in IC2D when you request the monitoring of the new machine involved (Monitoring menu, then "monitor new RMI host").

– the node just started has no active object running in it. Drag and drop one of the renderers, and check that the node is now taking place in the computation :

- spin the scene to trigger a new rendering
- see the topology

** if you feel uncomfortable with the automatic layout, switch to manual using the "manual layout" option (right click on the World panel). You can then reorganize the layout of the machines.*

– to fully distribute the computation, start several nodes (you need 2 more) and drag–and drop renderers in them.

Depending on the machines you have, the complexity of the image, look for the most efficient configuration.

7. have a look at the source code for the main classes of this application :

```
org.objectweb.proactive.examples.c3d.C3DUser.java
```

```
org.objectweb.proactive.examples.c3d.C3DRenderingEngine.java
```

```
org.objectweb.proactive.examples.c3d.C3DDispatcher.java
```

look at the method `public void processRotate(org.objectweb.proactive.Body body, String methodName, Request r)` that handles election of the next action to undertake.

2.3. Migration of active objects

ProActive allows the transparent migration of objects between virtual machines.

A nice visual example is the [penguin's one](#).

Mobile agents

This example shows a set of [mobile agents](#) moving around while still communicating with their base and with each other. It also features the capability to move a swing window between screens while moving an agent from one JVM to the other.

1. start the penguin application

using the `penguin` script.

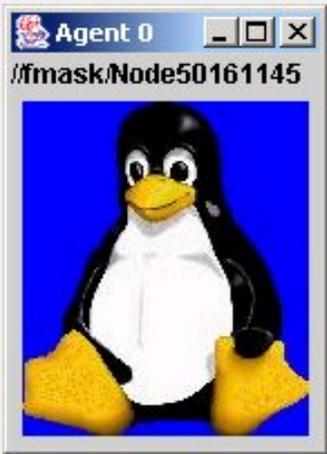
2. start IC2D to see what is going on

using the `ic2d` script

acquire the machines you have started nodes on

3. add an agent

– on the Advanced Penguin Controller window : button "add agent"



an agent is materialized by a picture in a java window.

– select it, and press button "start"

– observe that the active object is moving between the machines, and that the penguin window disappears and reappears on the screen associated with the new JVM.

4. add several agents

after selecting them, use the buttons to :

- communicate with them ("chained calls")
- start, stop, resume them
- trigger a communication between them ("call another agent")

5. move the control window to another user

- start a node on a different computer, using another screen and keyboard
- monitor the corresponding JVM with IC2D
- drag-and-drop the active object "AdvancedPenguinController" with IC2D into the newly created JVM : the control window will appear on the other computer and its user can now control the penguins application.
- still with IC2D, doing a drag-and-drop back to the original JVM, you will be able to get back the window, and control yourself the application.

3.0. SPMD Programming

OO SPMD on a Jacobi example

1. Execution and first glance at the Jacobi code

1.1 Source files: ProActive/src/org/objectweb/proactive/examples/jacobi

The Jacobi example is made of two Java classes:

- Jacobi.java: the main class
- SubMatrix.java: the class implementing the SPMD code

Have a first quick look at the code, especially the Jacobi class, looking for the strings "ProActive", "Nodes", "newSPMDGroup".

The last instruction of the class:

```
matrix.compute();
```

is an asynchronous group call. It sends a request to all active objects in the SPMD group, triggering computations in all the SubMatrix.

We will get to the class SubMatrix.java later on.

1.2 Execution

ProActive examples come with scripts to easily launch the execution under both Unix and Windows.

For Jacobi, launch:

```
ProActive/scripts/unix/jacobi.sh
```

or

```
ProActive/scripts/windows/jacobi.bat
```

The computation stops after minimal difference is reached between two iterations (constant MINDIFF in class Jacobi.java), or after a fixed number of iteration (constant ITERATIONS in class Jacobi.java).

The provided script, using an XML descriptor, creates 4 JVMs on the current machine. The Jacobi class creates an SPMD group of 9 Active Objects; 2 or 3 AOs per JVM.

Look at the traces on the console upon starting the script; in the current case, remember that all JVMs and AOs send output to the same console. More specifically, understand the following:

- "Created a new registry on port 1099"
- "Reading deployment descriptor ... Matrix.xml "
- "created VirtualNode"
- "**** Starting jvm on"
- "ClassFileServer is reading resources from classpath"
- "Detected an existing RMI Registry on port 1099"

```
- "Generating class : ... jacobi.Stub_SubMatrix "
- "ClassServer sent class ... jacobi.Stub_SubMatrix successfully"
```

You can start IC2D (script ic2d.sh or ic2d.bat) in order to visualize the JVMs and the Active Objects. Just activate the "Monitoring a new host" in the "Monitoring" menu at the top left.

To stop the Jacobi computation and all the associated AOs, and JVMs, just ^C in the window where you started the Jacobi script.

2. Modification and compilation

2.1 Source modification

Do a simple source modification, for instance changing the values of the constants MINDIFF (0.00000001 for ex) and ITERATIONS in class Jacobi.java.

Caveat: Be careful, due to a shortcoming of the Java make system (ant), make sure to also touch the class SubMatrix.java that uses the constants.

2.2 Compilation

ProActive distribution comes with scripts to easily recompile the provided examples:

```
linux>ProActive/compile/build
```

or

```
windows>ProActive/compile/build.bat
```

Several targets are provided (start build without arguments to obtain them). In order to recompile the Jacobi, just start the target that recompile all the examples:

```
build examples
```

2 source files must appear as being recompiled.

Following the recompilation, rerun the examples as explained in section 1.2 above, and observe the differences.

3. Detailed understanding of the OO SPMD Jacobi

3.1 Structure of the code

Within the class SubMatrix.java the following methods correspond to a standard Jacobi implementation, and are not specific to ProActive:

```
- internalCompute ()
- borderCompute ()
- exchange ()
- buildFakeBorder (int size)
- buildNorthBorder ()
- buildSouthBorder ()
- buildWestBorder ()
- buildEastBorder ()
- stop ()
```

The methods on which asynchronous remote method invocations take place

are:

- `sendBordersToNeighbors ()`
- `setNorthBorder (double[] border)`
- `setSouthBorder (double[] border)`
- `setWestBorder (double[] border)`
- `setEastBorder (double[] border)`

The first one sends to the appropriate neighbors the appropriate values, calling `set*Border()` methods asynchronously. Upon execution by the AO, the methods `set*Border()` memorize locally the values being received.

Notice that all those communication methods are made of purely functional Java code, without any code to the ProActive API.

On the contrary, the followings are ProActive related aspects:

- `buildNeighborhood ()`
- `compute ()`
- `loop ()`

We will detail them in the next section.

Note: the classes managing topologies are still under development. In the next release, the repetitive and tedious topology related instructions (e.g. methods `buildNeighborhood`) won't have to be written explicitly by the user, whatever the topology (2D, 3D).

3.2 OO SPMD behavior

Let us detail the OO SPMD techniques and ProActive related methods.

First of all, look for the definition and use of the attribute `"asyncRefToMe"`. Using the primitive `"getStubOnThis()"`, it provides a reference to the current active object ****on which method calls are asynchronous****. It permits the AO to send requests to itself.

For instance in

```
this.asyncRefToMe.loop();
```

Notice the absence of classical loop. The method `"loop()"` is indeed asynchronously called from itself; it is not really recursivity since it does not have the drawback of the stack growing. It features an important advantage: the AO will remain reactive to other calls being sent to him. Moreover, it eases reuse since it is not necessary to explicitly encode within the main SPMD loop all the messages that have to be taken into account. It also facilitates composition since services can be called by activities outside the SPMD group, they will be automatically executed by the FIFO service of the Active Object.

The method `"buildNeighborhood ()"` is called only once for initialization. Using a 2D topology (Plan), it constructs references to north, south, west, east neighbors — attributes with respective names. It also construct dynamically the group of neighbors. Starting from an empty group of type `SubMatrix`

```
this.neighbors = (SubMatrix) ProActiveGroup.newGroup
(SubMatrix.class.getName());
```

such typed view of the group is used to get the group view: `Group neighborsGroup = ProActiveGroup.getGroup(this.neighbors)`; Then, the appropriate neighbors are added dynamically in the group, e.g.:

```
neighborsGroup.add(this.north);
```

Again, the classes managing topologies will permit to simplify this code.

3.3 Adding a Method barrier for a step by step execution

Let say we would like to control step by step the execution of the SPMD code. We will add a barrier in the `SubMatrix.java`, and control the barrier from input in the `Jacobi.java` class.

In class `SubMatrix.java`, add a Method `barrier()` of the form:

```
String[] st= new String[1];

st[0]="keepOnGoing";

ProSPMD.barrier(st);
```

Do not forget to define the `keepOnGoing()` method that indeed can return void, and just be empty. Find the appropriate place to call the `barrier()` Method in the `loop()` Method.

In class `Jacobi.java`, just after the `compute()` Method, add an infinite loop that, upon a user's return key pressed, calls the method `keepOnGoing()` on the SPMD group "matrix". Here are samples of the code:

```
while (true) {

printMessageAndWait();

matrix.keepOnGoing();

}

...

private static void printMessageAndWait() {

java.io.BufferedReader d = new java.io.BufferedReader(

new java.io.InputStreamReader(System.in));

System.out.println(" --> Press return key to continue");

System.out.println(" or Ctrl c to stop.");

try {

d.readLine();

} catch (Exception e) {

}

}

}
```

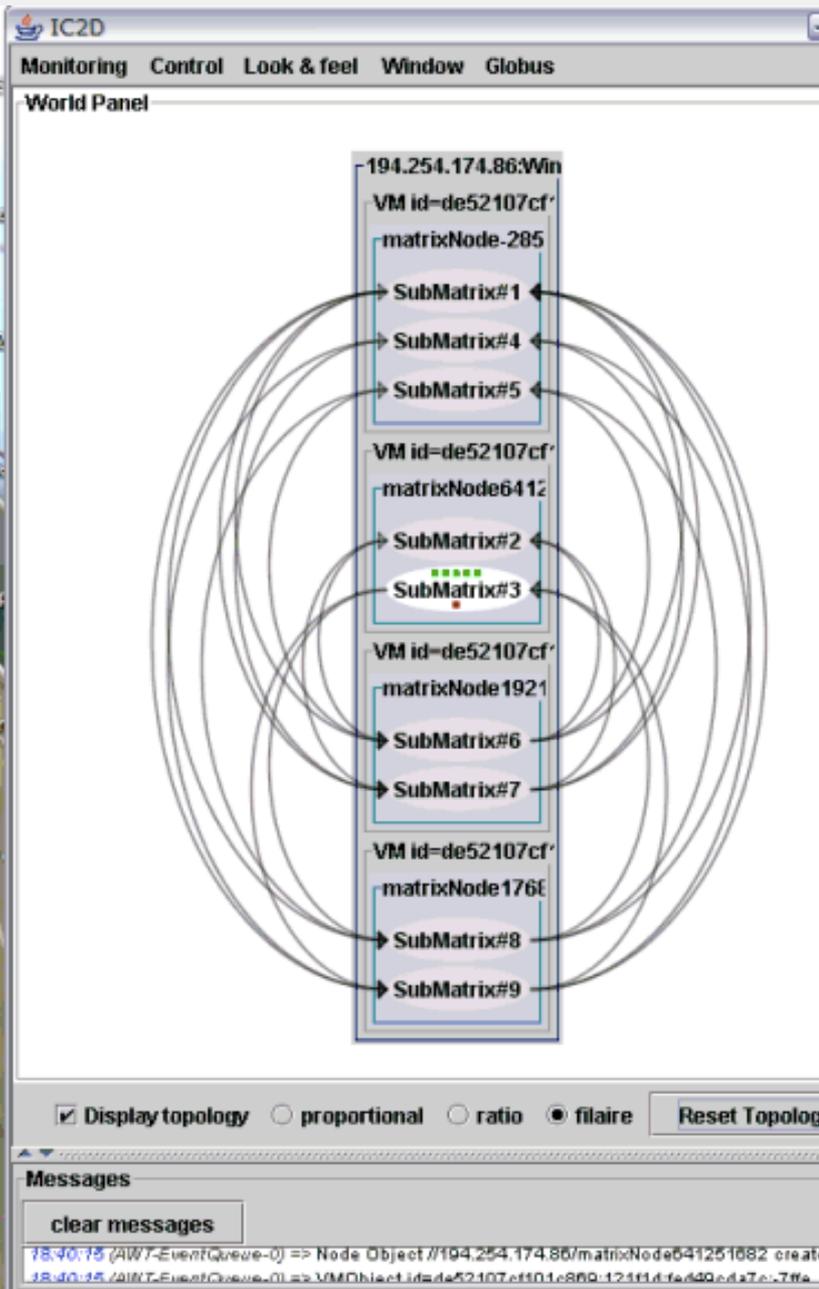
Recompile, and execute the code. Each iteration needs to be activated by hitting the return key in the shell window where `Jacobi` was launched. Start `IC2D` (`./ic2d.sh` or `ic2d.bat`), and visualize the communications as you control them. Use the "Reset Topology" button to clear communication arcs. The green and red dots indicate the pending requests.

You can imagine and test other modifications to the `Jacobi` code.

```

WINDOWS\System32\cmd.exe - jacobi.bat
is ClassFileServer is reading resources from classpath
abled
abled
abled
abled
abled
abled
d an existing RMI Registry on port 1099
ve Security Policy (proactive.runtime.security) not set. Ru
ve Security Policy (proactive.runtime.security) not set. Ru
e started at rmi://194.254.174.86/PA_JVM-882970219_194.254.1
ve Security Policy (proactive.runtime.security) not set. Ru
ve Security Policy (proactive.runtime.security) not set. Ru
e started at rmi://194.254.174.86/PA_JVM-1216932264_194.254.
e started at rmi://194.254.174.86/PA_JVM-1240017198_194.254.
e started at rmi://194.254.174.86/PA_JVM-1166914907_194.254.
54.174.86/matrixNode-285238994 successfully bound in regist
pping VirtualNode matrixNode with Node: //194.254.174.86/ma
pping VirtualNode matrixNode with Node: //194.254.174.86/ma
54.174.86/matrixNode1768935384 successfully bound in regist
pping VirtualNode matrixNode with Node: //194.254.174.86/ma
54.174.86/matrixNode641251682 successfully bound in regist
pping VirtualNode matrixNode with Node: //194.254.174.86/ma
54.174.86/matrixNode1921326597 successfully bound in regist
ing class : pa.stub.org.objectweb.proactive.examples.jacobi
rver sent class pa.stub.org.objectweb.proactive.examples.ja
rver sent class pa.stub.org.objectweb.proactive.examples.ja
rver sent class pa.stub.org.objectweb.proactive.examples.ja
rver sent class pa.stub.org.objectweb.proactive.examples.ja
ess return key to continue
Ctrl c to stop.
ing class : pa.stub.org.objectweb.proactive.examples.jacobi
ing class : pa.stub.org.objectweb.proactive.examples.jacobi
ing class : pa.stub.org.objectweb.proactive.examples.jacobi
ing class : pa.stub.org.objectweb.proactive.examples.jacobi
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.
ess return key to continue
Ctrl c to stop.

```



3.4 Understanding various different kind of barriers

The group of neighbors built above is important wrt synchronization. Below in method "loop()", an efficient barrier is achieved only using the direct neighbors:

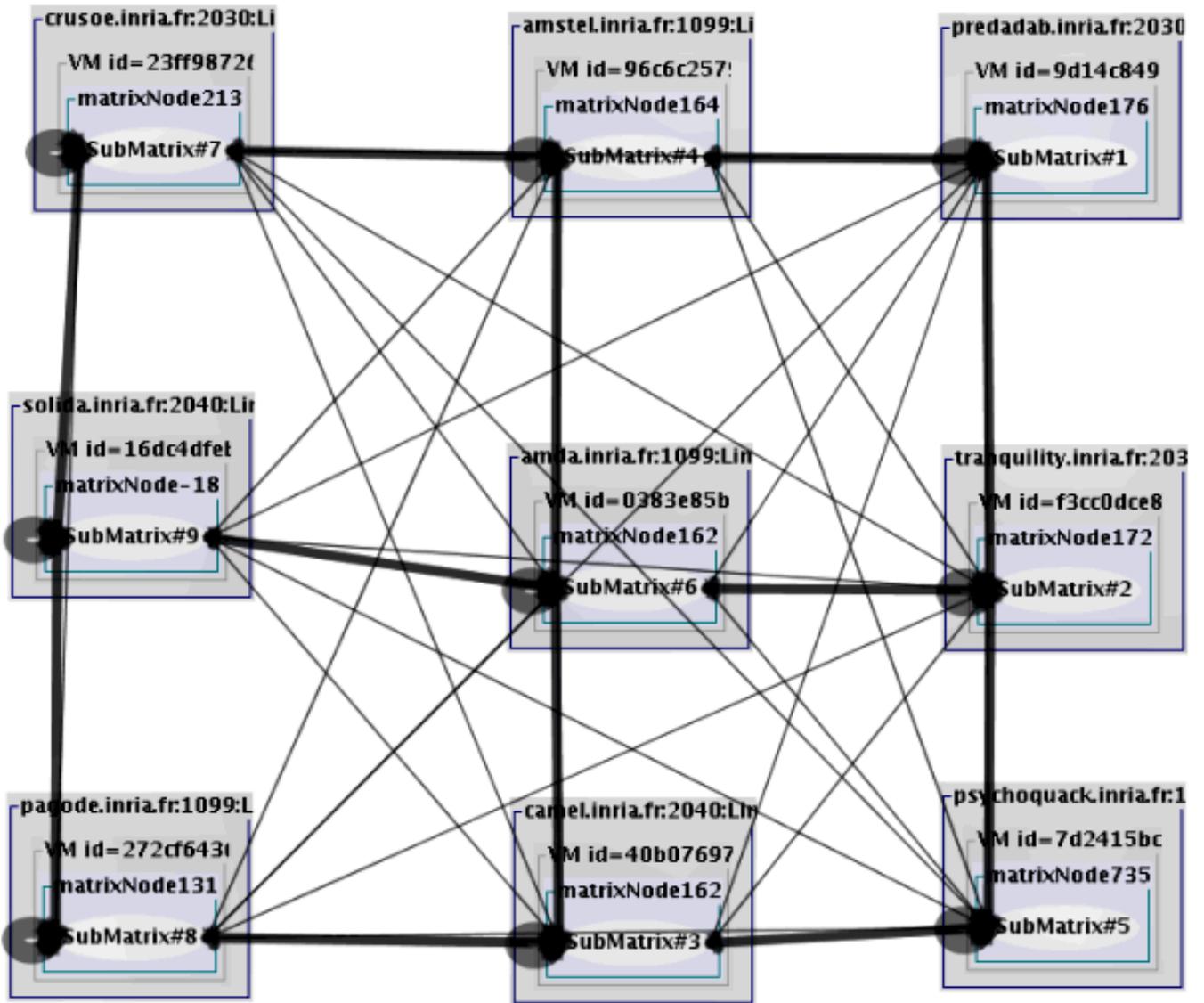
```
ProSPMD.barrier("SynchronizationWithNeighbors"+ this.iterationsToStop, this.neighbors);
```

This barrier takes as a parameter the group to synchronize with: it will be passed only when the 4 neighbors in the current 2D example have reached the same point. Adding the rank of the current iteration allows to have a unique identifier for each instance of the barrier.

Try to change the barrier instruction to a total barrier:

```
ProSPMD.barrier("SynchronizationWithNeighbors"+ this.iterationsToStop);
```

Then recompile and execute again. Using IC2D observe that many more communications are necessary.



Display topology proportional ratio filaire Monitoring enable

ges

...mml.server.UnicastServerRef dispatch(UnicastServerRef.java:246)



Display topology proportional ratio filaire Monitoring enable

In order to get details and documentation on Groups and OO SPMD, have a look at:

[ProActive/src/org/objectweb/proactive/doc-files/](#)

[TypedGroupCommunication.html](#)

[OOSPMO.html](#)

4. Virtual Nodes and Deployment descriptors

4.1 Virtual Nodes

Get back to the source code of `Jacobi.java`, and understand where and how the Virtual Nodes and Nodes are being used.

4.2 XML Descriptors

The XML descriptor being used is:

[ProActive/descriptors/Matrix.xml](#)

Look for and understand the following definitions:

- Virtual Node Definition
- Mapping of Virtual Nodes to JVM
- JVM Definition
- Process Definition

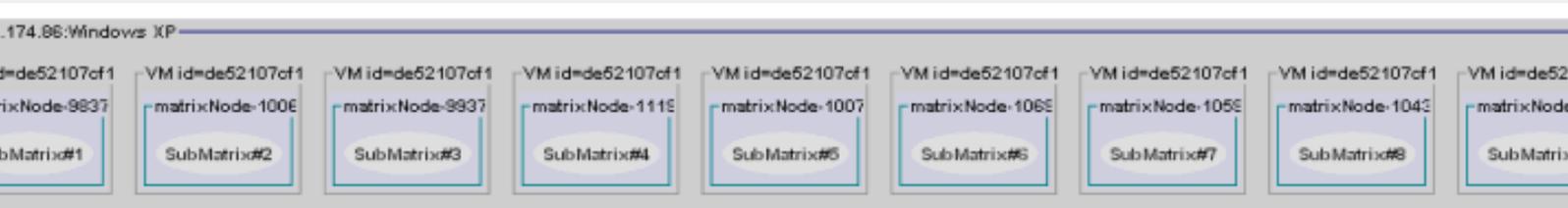
A detailed presentation of XML descriptors is available at:

ProActive/docs/api/index.html

entry 9. XML Deployment Descriptors

4.3 Changing the descriptor

Edit the file `Matrix.xml` in order to change the number of JVMs being used. For instance, if your machine is powerful enough, start 9 JVMs, in order to have a single `SubMatrix` per JVM.



You do not need to recompile, just restart the execution. Use IC2D to visualize the differences in the configuration.

5. Execution on several machines and Clusters

5.1 Execution on several machines in the room

Associate with several persons or use several machines, and modify the file

ProActive/examples/descriptors/Matrix.xml

in order to launch the Jacobi computation on several machines. You can use IC2D to visualize the machines and the JVMs being launched on them.

5.2 Execution on Clusters

If you have access to your own clusters, configure the XML descriptor to launch the Jacobi on them, using the appropriate protocol:

ssh, LSF, PBS, Globus, etc.

Have a look at

ProActive/docs/api/org/objectweb/proactive/doc-files/Descriptor.html

to get the format of the XML descriptor for each of the supported protocols.

3. Conclusion

This tour was intended to guide you through an overview of ProActive.

You should now be able to start programming with ProActive, and you should also have an idea of the capabilities of the library.

We hope that you liked it and we thank you for your interest in ProActive.

Further information can be found on the website, and [suggestions](#) are welcome.

```
import org.objectweb.proactive.ProActive;

public class Hello {
    private String name;
    private String hi = "Hello world";
    private java.text.DateFormat dateFormat = new java.text.SimpleDateFormat(
        "dd/MM/yyyy HH:mm:ss");

    public Hello() {
    }

    public Hello(String name) {
        this.name = name;
    }

    public String sayHello() {
        return hi + " at " + dateFormat.format(new java.util.Date()) +
            " from node : " + ProActive.getBodyOnThis().getNodeURL();
    }

    public static void main(String[] args) {
        // Registers it with an URL
        try {
            // Creates an active instance of class HelloServer on the local node
            Hello hello = (Hello) org.objectweb.proactive.ProActive.newActive(Hello.class.getName(),
                new Object[] { "remote" });
            java.net.InetAddress localhost = java.net.InetAddress.getLocalHost();
            org.objectweb.proactive.ProActive.register(hello,
                "/" + localhost.getHostName() + "/Hello");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}

public class HelloClient {
    public static void main(String[] args) {
        Hello myServer;
        String message;
        try {
            // checks for the server's URL
            if (args.length == 0) {
                // There is no url to the server, so create an active server within this VM
                myServer = (Hello) org.objectweb.proactive.ProActive.newActive(Hello.class.getName(),
                    new Object[] { "local" });
            } else {
                // Lookups the server object
                System.out.println("Using server located on " + args[0]);
                myServer = (Hello) org.objectweb.proactive.ProActive.lookupActive(Hello.class.getName(),
                    args[0]);
            }

            // Invokes a remote method on this object to get the message
            message = myServer.sayHello();
            // Prints out the message
            System.out.println("The message is : " + message);
        } catch (Exception e) {
    }
}
```

```

System.err.println("Could not reach/create server object");
e.printStackTrace();
System.exit(1);
}
}
}

```

```

package org.objectweb.proactive.examples.hello;

```

```

import org.objectweb.proactive.Body;
import org.objectweb.proactive.EndActive;
import org.objectweb.proactive.InitActive;
import org.objectweb.proactive.ProActive;

```

```

public class InitializedHello extends Hello implements InitActive, EndActive {

```

```

    /**
     * Constructor for InitializedHello.
     */

```

```

    public InitializedHello() {
    }

```

```

    /**
     * Constructor for InitializedHello.
     * @param name
     */

```

```

    public InitializedHello(String name) {
        super(name);
    }

```

```

    /**
     * @see org.objectweb.proactive.InitActive#initActivity(Body)
     * This is the place where to make initialization before the object
     * starts its activity
     */

```

```

    public void initActivity(Body body) {
        System.out.println("I am about to start my activity");
    }

```

```

    /**
     * @see org.objectweb.proactive.EndActive#endActivity(Body)
     * This is the place where to clean up or terminate things after the
     * object has finished its activity
     */

```

```

    public void endActivity(Body body) {
        System.out.println("I have finished my activity");
    }

```

```

    /**
     * this method will end the activity of the active object
     */

```

```

    public void terminate() {
        // the termination of the activity is done through a call on the
        // terminate method of the body associated to the current active object
        ProActive.getBodyOnThis().terminate();
    }

```

```

    public static void main(String[] args) {

```

```

        // Registers it with an URL

```

```

        try {

```

```

            // Creates an active instance of class HelloServer on the local node

```

```

            InitializedHello hello = (InitializedHello) org.objectweb.proactive.ProActive.newActive(InitializedHello.class.getName()

```

```

                + ".class",
                new Object[] { "remote" });

```

```

            java.net.InetAddress localhost = java.net.InetAddress.getLocalHost();

```

```

            org.objectweb.proactive.ProActive.register(hello,

```

```

                "://" + localhost.getHostName() + "/Hello");

```

```

        } catch (Exception e) {

```

```

            System.err.println("Error: " + e.getMessage());

```

```

            e.printStackTrace();

```

```

        }

```

```

    }

```

```

}

```

```
package org.objectweb.proactive.examples.hello;
```

```
public class InitializedHelloClient {
    public static void main(String[] args) {
        InitializedHello myServer;
        String message;
        try {
            // checks for the server's URL
            if (args.length == 0) {
                // There is no url to the server, so create an active server within this VM
                myServer = (InitializedHello) org.objectweb.proactive.ProActive.newActive(InitializedHello.class.getName(),
                    new Object[] { "local" });
            } else {
                // Lookups the server object
                System.out.println("Using server located on " + args[0]);
                myServer = (InitializedHello) org.objectweb.proactive.ProActive.lookupActive(InitializedHello.class.getName(),
                    args[0]);
            }

            // Invokes a remote method on this object to get the message
            message = myServer.sayHello();
            // Prints out the message
            System.out.println("The message is : " + message);
            myServer.terminate();
        } catch (Exception e) {
            System.err.println("Could not reach/create server object");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

```
package org.objectweb.proactive.examples.hello;
```

```
import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.body.migration.MigrationException;
import org.objectweb.proactive.core.node.NodeException;
```

```
import java.io.Serializable;
```

```
// the object that will be migrated active has to be Serializable
public class MigratableHello extends InitializedHello implements Serializable {

    /**
     * Creates a new MigratableHello object.
     */
    public MigratableHello() {
    }

    /**
     * Creates a new MigratableHello object.
     *
     * @param name the name of the agent
     */

    // ProActive requires the active object to explicitly define (or redefine)
    // the constructors, so that they can be reified
    public MigratableHello(String name) {
        super(name);
    }

    /** factory for locally creating the active object
     * @param name the name of the agent
     * @return an instance of a ProActive active object of type MigratableHello
     */
    public static MigratableHello createMigratableHello(String name) {
        try {
            return (MigratableHello) ProActive.newActive(MigratableHello.class.getName(),
                new Object[] { name });
        } catch (ActiveObjectCreationException aoce) {
            System.out.println("creation of the active object failed");
        }
    }
}
```

```

        aoce.printStackTrace();
        return null;
    } catch (NodeException ne) {
        System.out.println("creation of default node failed");
        ne.printStackTrace();
        return null;
    }
}

/** method for migrating
 * @param destination_node destination node
 */
public void moveTo(String destination_node) {
    System.out.println("\n-----");
    System.out.println("starting migration to node : " + destination_node);
    System.out.println("...");
    try {
        // THIS MUST BE THE LAST CALL OF THE METHOD
        ProActive.migrateTo(destination_node);
    } catch (MigrationException me) {
        System.out.println("migration failed : " + me.toString());
    }
}
}
}

```

```
package org.objectweb.proactive.examples.hello;
```

```

public class MigratableHelloClient {

    /** entry point for the program
     * @param args destination nodes
     * for example :
     * rmi://localhost/node1 jini://localhost/node2*/
    public static void main(String[] args) { // instantiation-based creation of the active object
        MigratableHello migratable_hello = MigratableHello.createMigratableHello(
            "agent1");

        // check if the migratable_hello has been created
        if (migratable_hello != null) {
            // say hello
            System.out.println(migratable_hello.sayHello());
            // start moving the object around
            for (int i = 0; i < args.length; i++) {
                migratable_hello.moveTo(args[i]);
                System.out.println("received message : " +
                    migratable_hello.sayHello());
            }

            // possibly terminate the activity of the active object ...
            migratable_hello.terminate();
        } else {
            System.out.println("creation of the active object failed");
        }
    }
}
}

```

```
package org.objectweb.proactive.examples.hello;
```

```

/**
 * This class allows the creation of a graphical window
 * with a text field
 */
public class HelloFrame extends javax.swing.JFrame {
    private javax.swing.JLabel jLabel1;

    /** Creates new form HelloFrame */
    public HelloFrame(String text) {
        initComponents();
        setText(text);
    }
}

```

```

/** This method is called from within the constructor to
 * initialize the form.
 * It will perform the initialization of the frame
 */
private void initComponents() {
    jLabel1 = new javax.swing.JLabel();
    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent evt) {
            exitForm(evt);
        }
    });

    jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
    getContentPane().add(jLabel1, java.awt.BorderLayout.CENTER);
}

/** Kill the frame */
private void exitForm(java.awt.event.WindowEvent evt) {
    // System.exit(0); would kill the VM !
    dispose(); // this way, the active object agentFrameController stays alive
}

/**
 * sets the text of the label inside the frame
 */
private void setText(String text) {
    jLabel1.setText(text);
}
}

package org.objectweb.proactive.examples.hello;

import org.objectweb.proactive.ActiveObjectCreationException;
import org.objectweb.proactive.Body;
import org.objectweb.proactive.ProActive;
import org.objectweb.proactive.core.body.migration.Migratable;
import org.objectweb.proactive.core.node.NodeException;
import org.objectweb.proactive.ext.migration.MigrationStrategyManager;
import org.objectweb.proactive.ext.migration.MigrationStrategyManagerImpl;

/**
 * This class allows the "migration" of a graphical interface. A gui object is attached
 * to the current class, and the gui is removed before migration, thanks to the use
 * of a MigrationStrategyManager
 */
public class HelloFrameController extends MigratableHello {
    HelloFrame helloFrame;
    MigrationStrategyManager migrationStrategyManager;

    /**required empty constructor */
    public HelloFrameController() {
    }

    /**constructor */
    public HelloFrameController(String name) {
        super(name);
    }

    /**
     * This method attaches a migration strategy manager to the current active object.
     * The migration strategy manager will help to define which actions to take before
     * and after migrating
     */
    public void initActivity(Body body) {
        // add a migration strategy manager on the current active object
        migrationStrategyManager = new MigrationStrategyManagerImpl((Migratable) ProActive.getBodyOnThis());
        // specify what to do when the active object is about to migrate
        // the specified method is then invoked by reflection
        migrationStrategyManager.onDeparture("clean");
    }

    /** factory for locally creating the active object

```

```
* @param name the name of the agent
* @return an instance of a ProActive active object of type HelloFrameController
*
*/
```

```
public static HelloFrameController createHelloFrameController(String name) {
try {
// creates (and initialize) the active object
HelloFrameController obj =
(HelloFrameController) ProActive.newActive(HelloFrameController.class.getName(), new Object[] { name });
return obj;
} catch (ActiveObjectCreationException aoce) {
System.out.println("creation of the active object failed");
aoce.printStackTrace();
return null;
} catch (NodeException ne) {
System.out.println("creation of default node failed");
ne.printStackTrace();
return null;
}
}
```

```
public String sayHello() {
if (helloFrame == null) {
helloFrame = new HelloFrame("Hello from " + ProActive.getBodyOnThis().getNodeURL());
helloFrame.show();
}
return "Hello from " + ProActive.getBodyOnThis().getNodeURL();
}
```

```
public void clean() {
System.out.println("killing frame");
helloFrame.dispose();
helloFrame = null;
System.out.println("frame is killed");
}
}
```
