ProActive

The Java library for **Parallel, Distributed, Concurrent** computing with **Security** and **Mobility**

version 2.0 (April 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 functionnalities 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

Table of Contents

0. Installation and setup

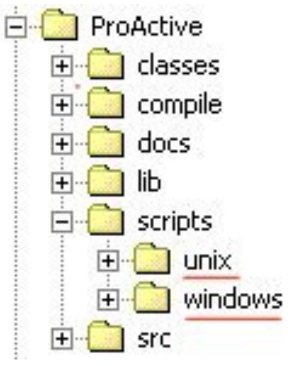
Follow the instructions for downloading and installing ProActive.

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

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



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".

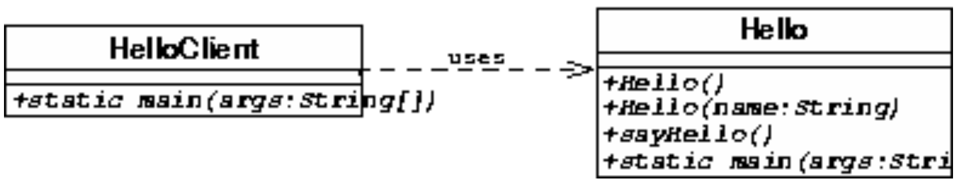
The program that we will develop is a kind of "helloworld" example. We will increase the complexity of the example, so that you familiarize yourself with different features of ProActive.

- 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 mentionned 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

```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 : " + 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:

```
> java org.objectweb.proactive.examples.hello.Client &
```

Hello World from another VM on the same host

Starting the server

Just start the `main` method in the `Hello` class.

```
> java org.objectweb.proactive.examples.hello.Hello &
```

Launching the client

```
> java 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.

```
remoteHost> java org.objectweb.proactive.examples.hello.Hello &
```

Launching the client

Log on to the client Host, and launch the client

```
clientHost> java 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.

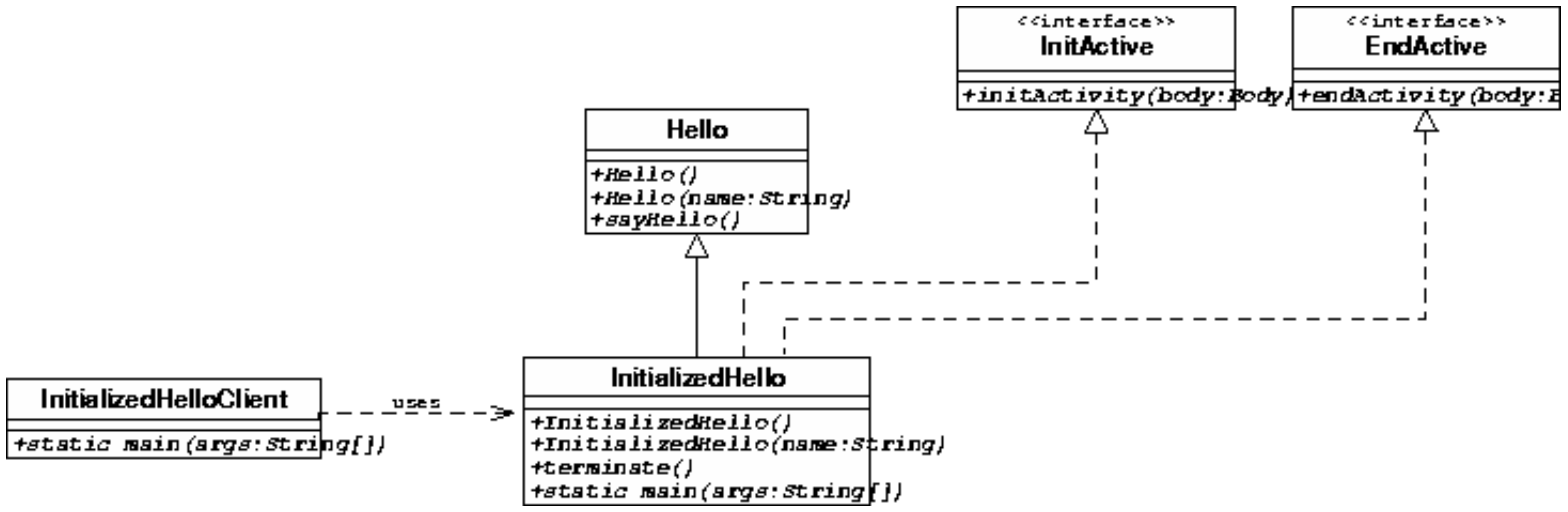
When instanciated, the activity of an object is automatically started, but it will first do what is written in the initActivity method.

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

Design of the application

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

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



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();
}
```

InitializedHelloClient

Code is here.

The only differences from the HelloClient of the previous example is the class instantiated, which is now InitializedHello (and not Hello), and you will add a call to hello.terminate(). The source code is here.

Execution

Execution is similar to the previous example; just use the InitializedHelloClient client class.

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

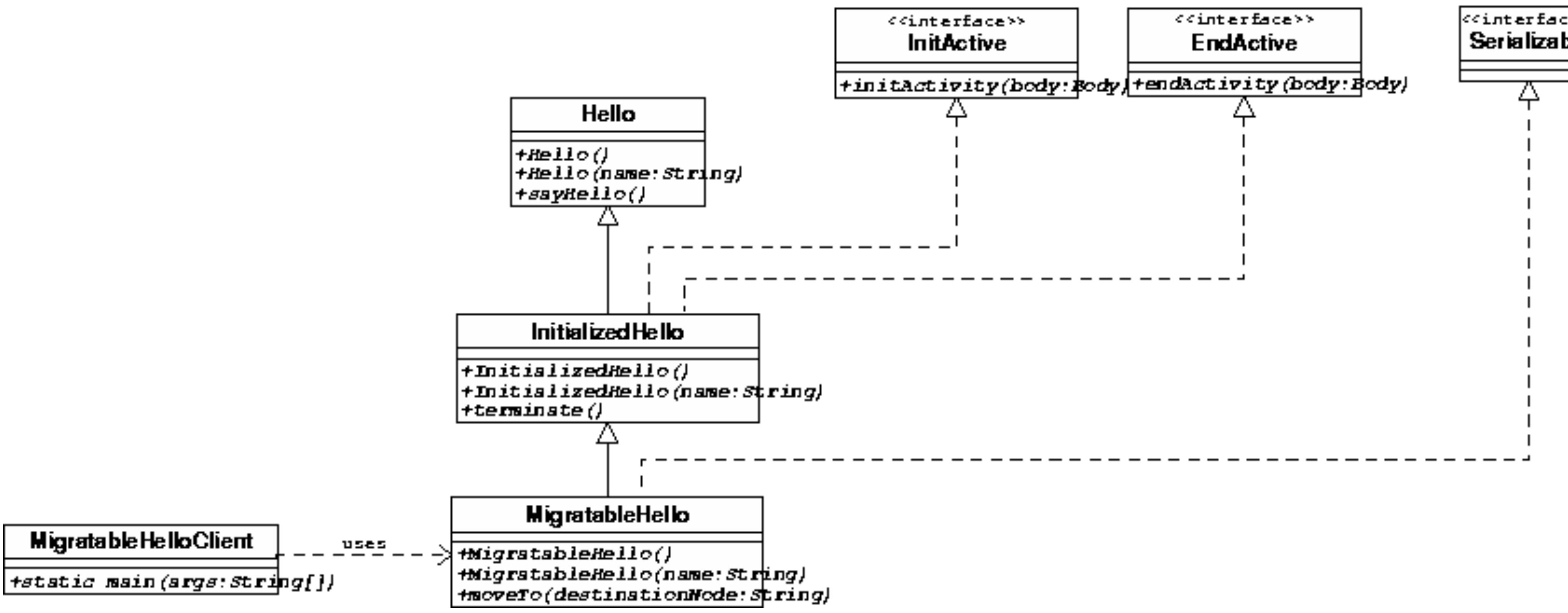
We want to further enhance InitializedHello it by making migratable : we'd like to be able to move it across virtual machines.

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.

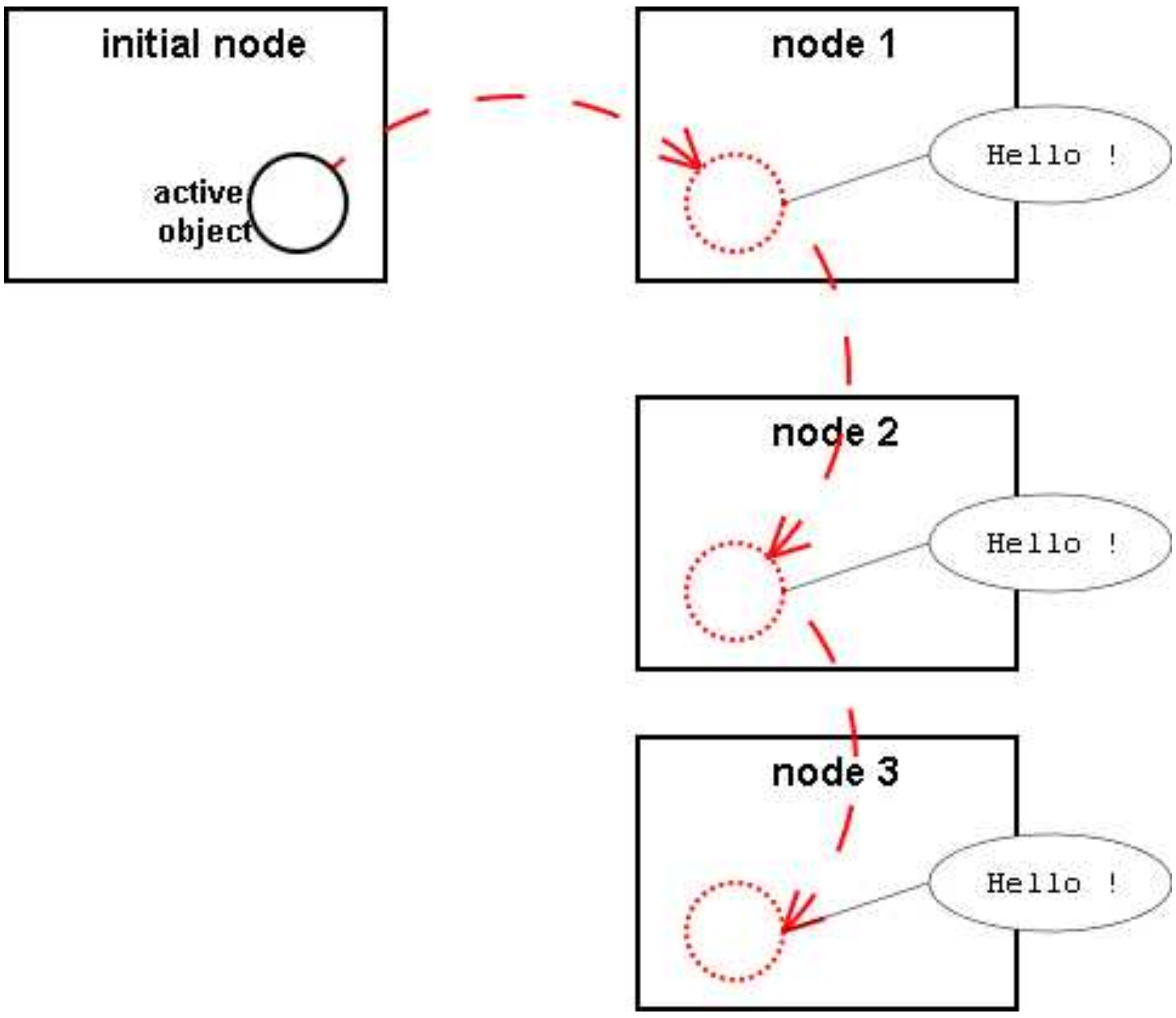
1.3.4. Execution

− start several nodes using the `startnode` script.

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

− don't forget to set the security permissions

− 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.

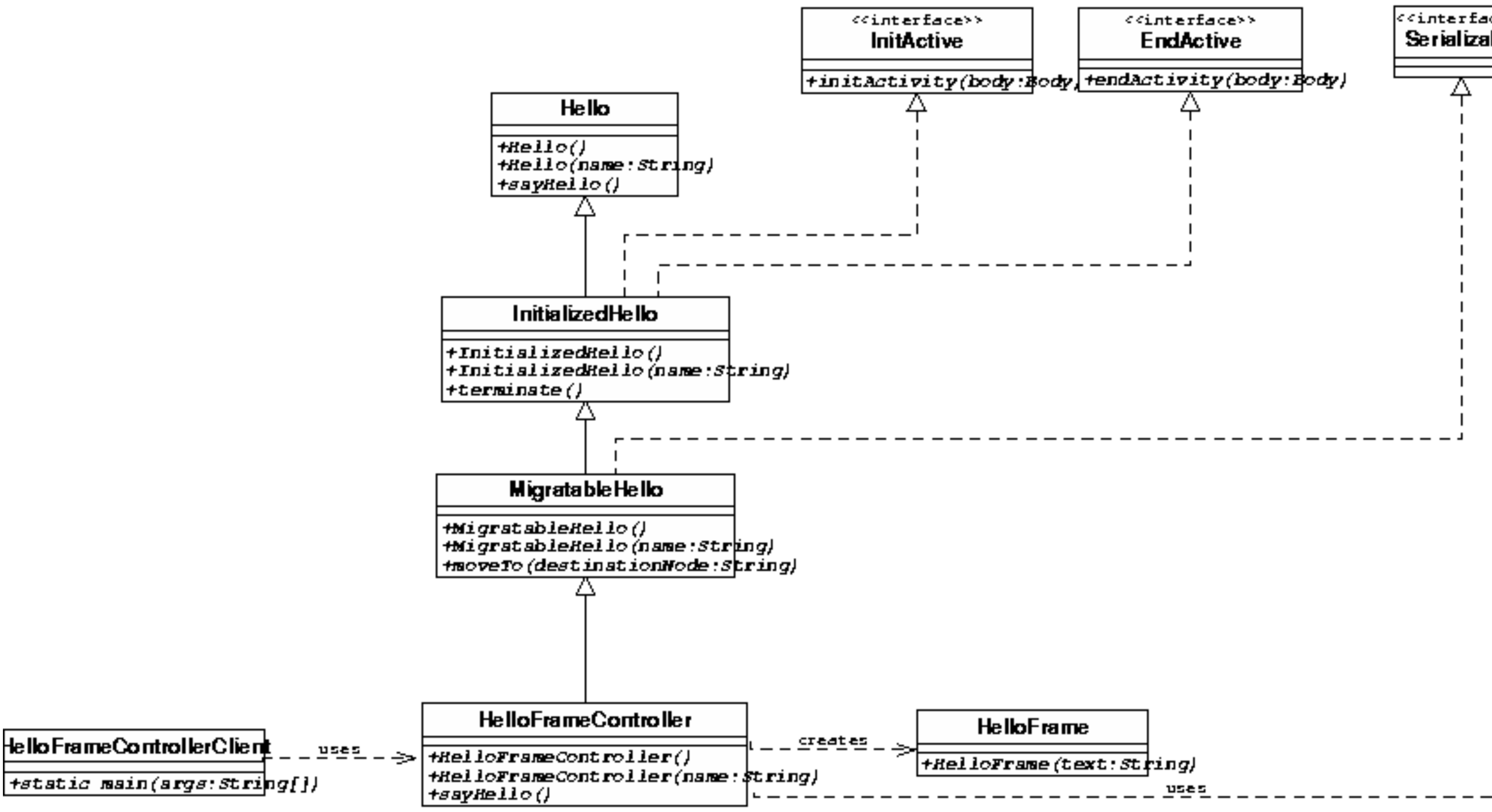
1.4. migration of graphical interfaces

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 desactivation 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");
}
```

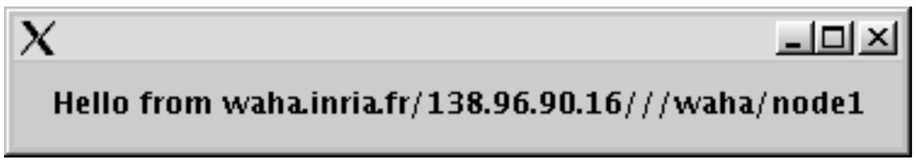
HelloFrame

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.
- don't forget to set the security permissions
- 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
```



Hello from waha.inria.fr/138.96.90.16///waha/node1

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

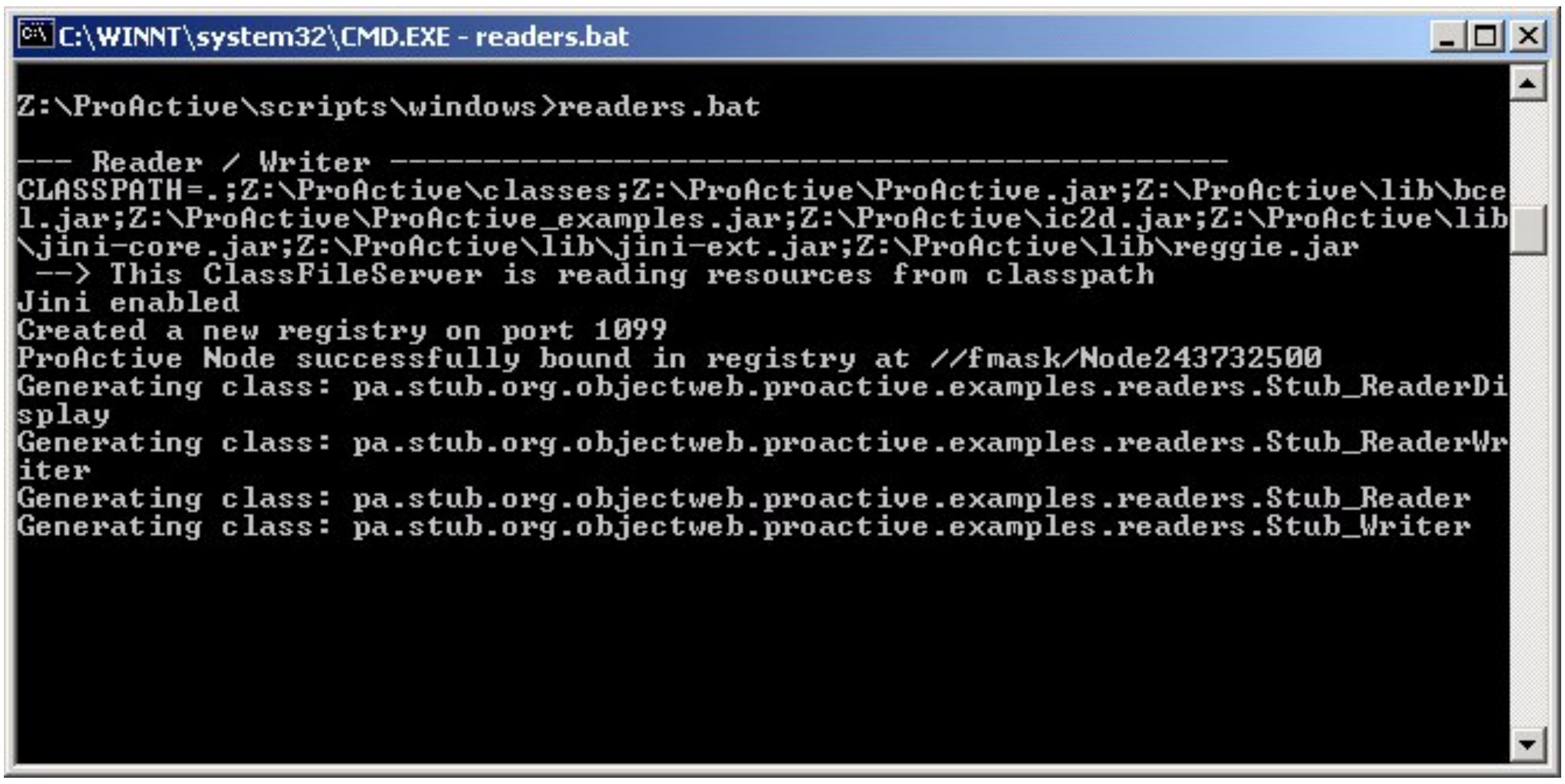
The readers−writers

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

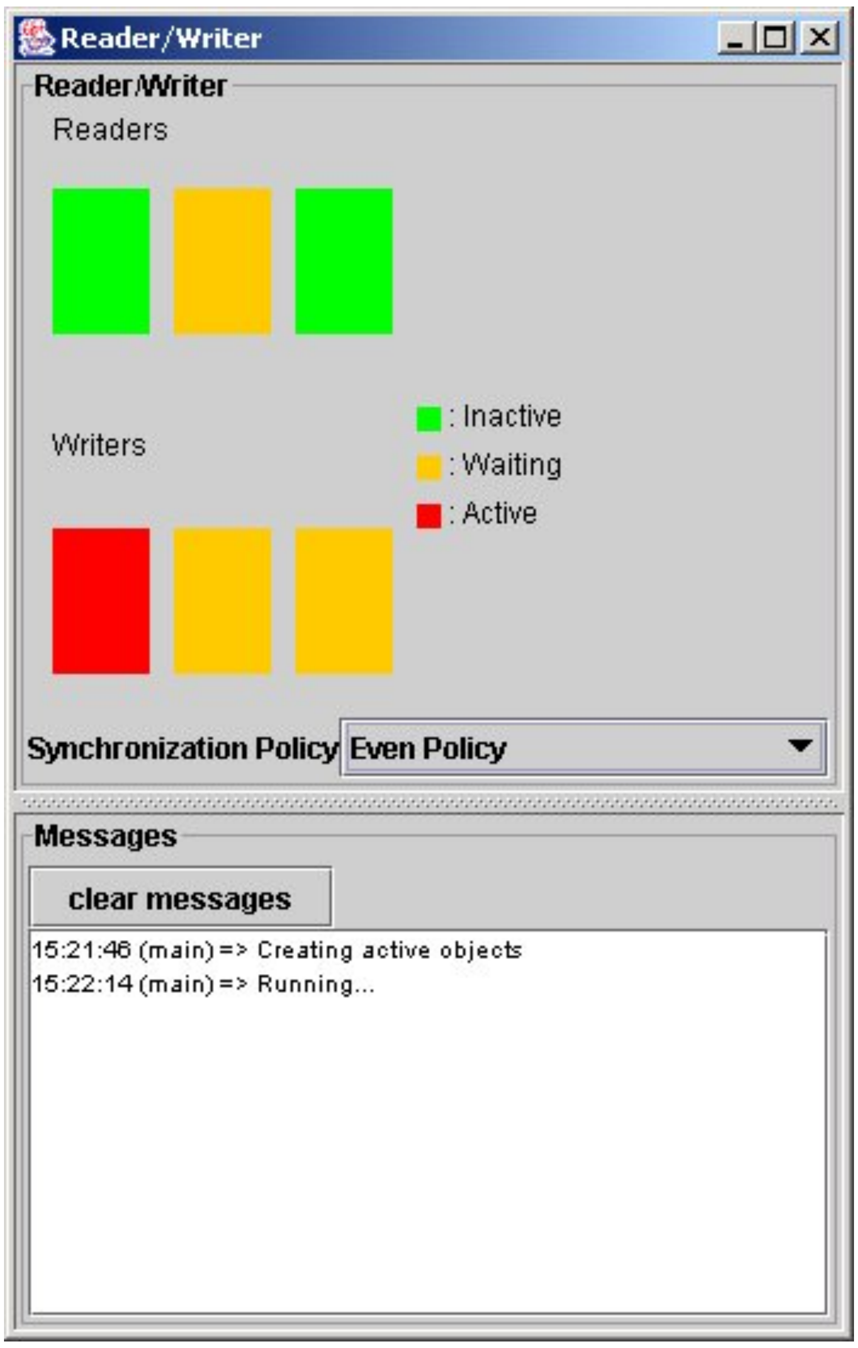
using the readers script



```
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.



Synchronization Policy: Even Policy

```
15:21:46 (main) => Creating active objects
15:22:14 (main) => Running...
```

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 `MyRequestFilterm` 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

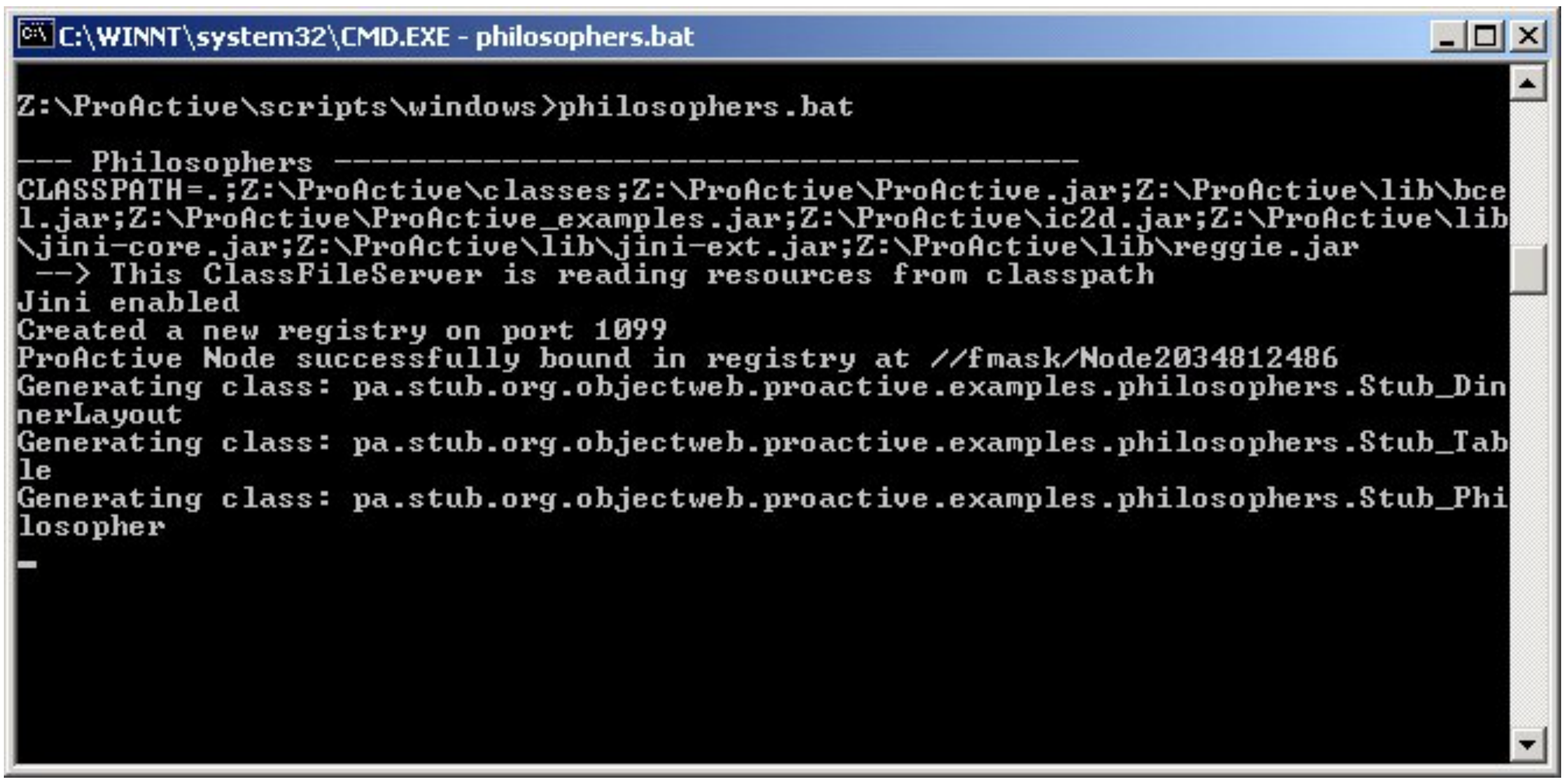
The dining philosophers

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

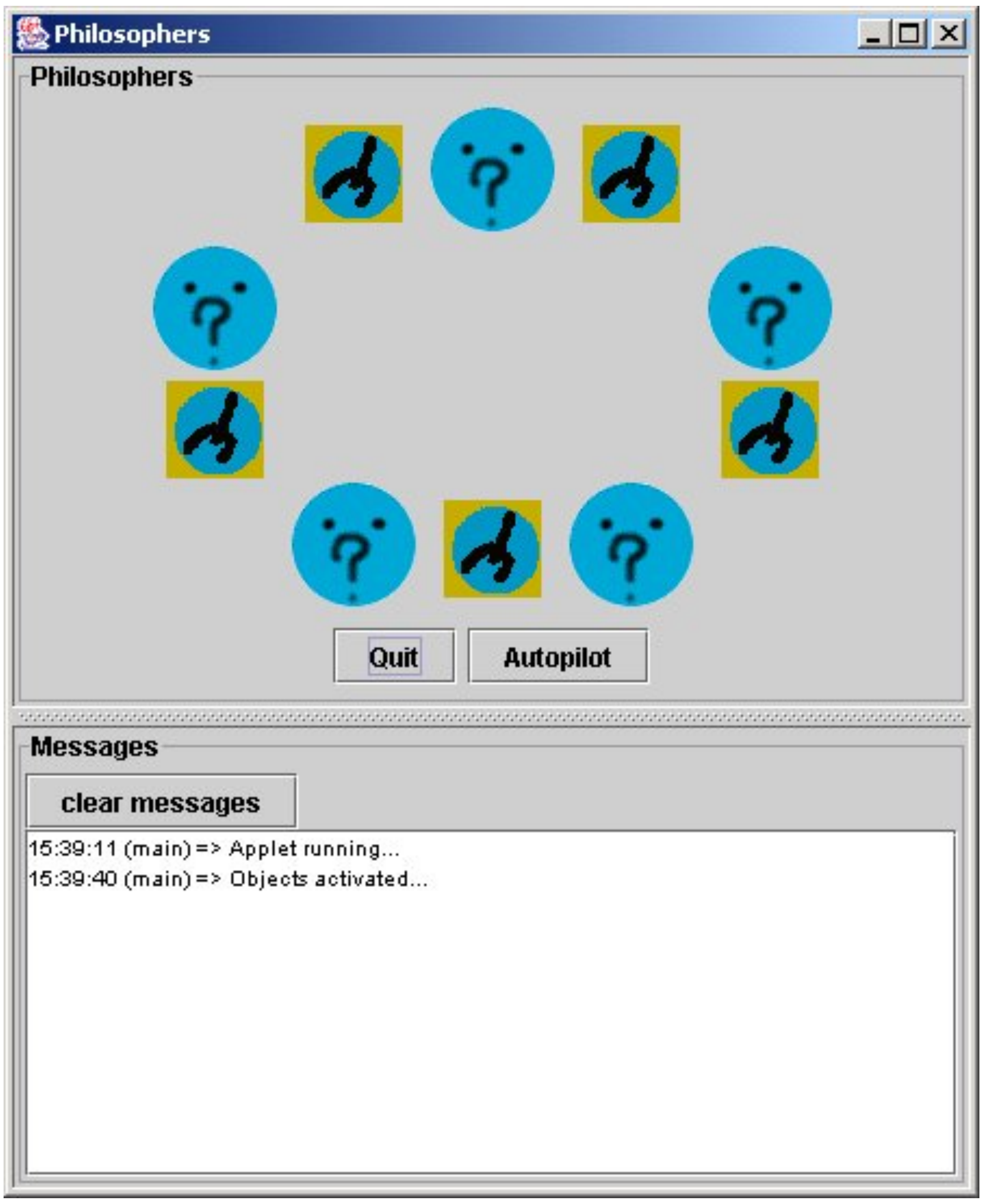
with philosophers.sh or philosophers.bat



```
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
_
```

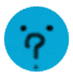
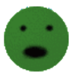
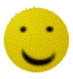
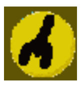
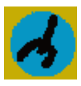
ProActive creates a new node and instantiates the active objects of the application : DinnerLayout, Table, and Philosopher



```
15:39:11 (main) => Applet running...
15:39:40 (main) => Objects activated...
```

the GUI is started.

2. understand the color codes

Philosophers		*philosophing*
		*hungry, wants the fork !*
		*eating*
Forks		*taken*
		*free*

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

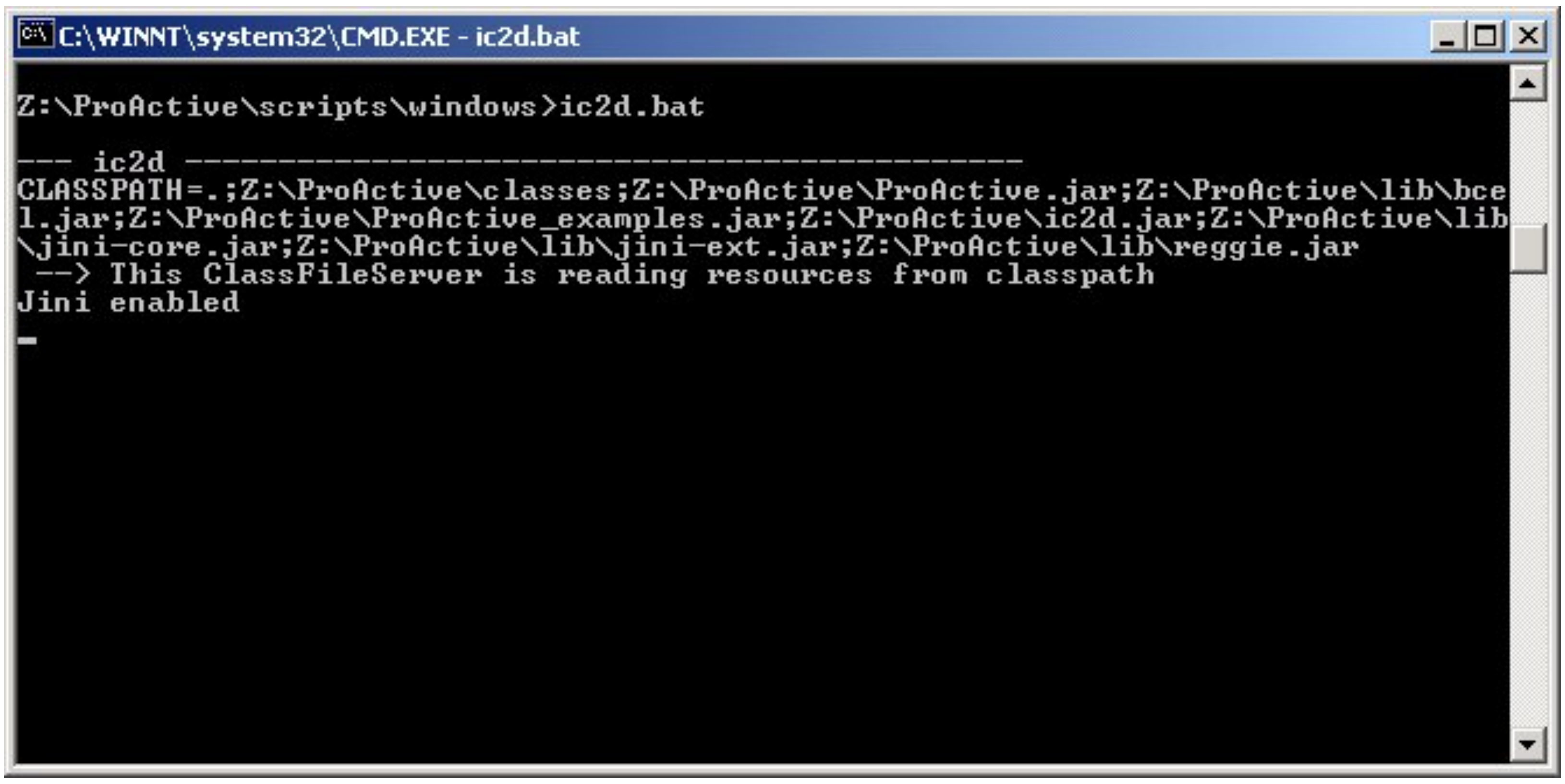
Test that there are no deadlocks!

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.

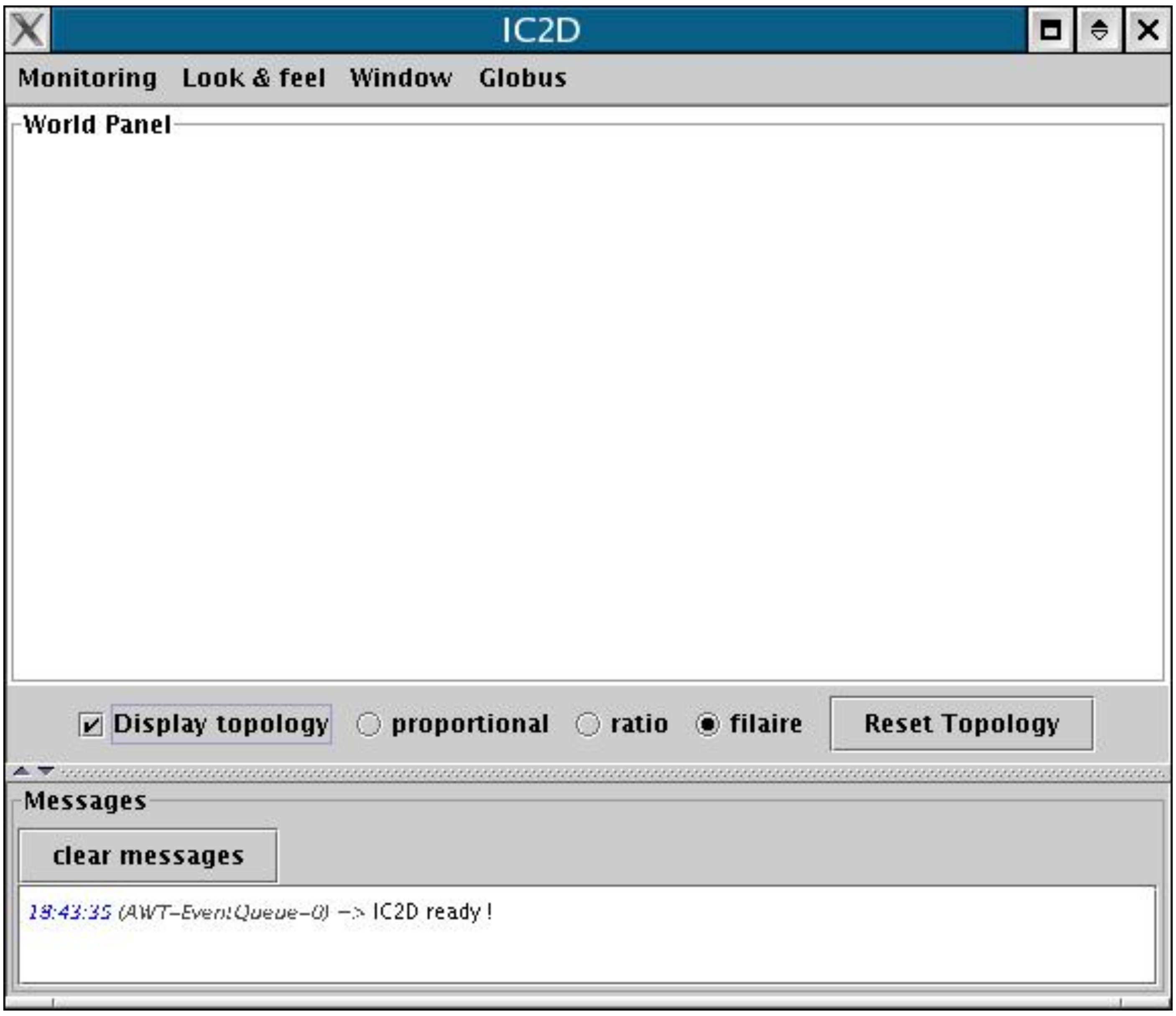
− being in the autopilot mode, start the IC2D visualization application (using ic2d.sh or ic2d.bat)



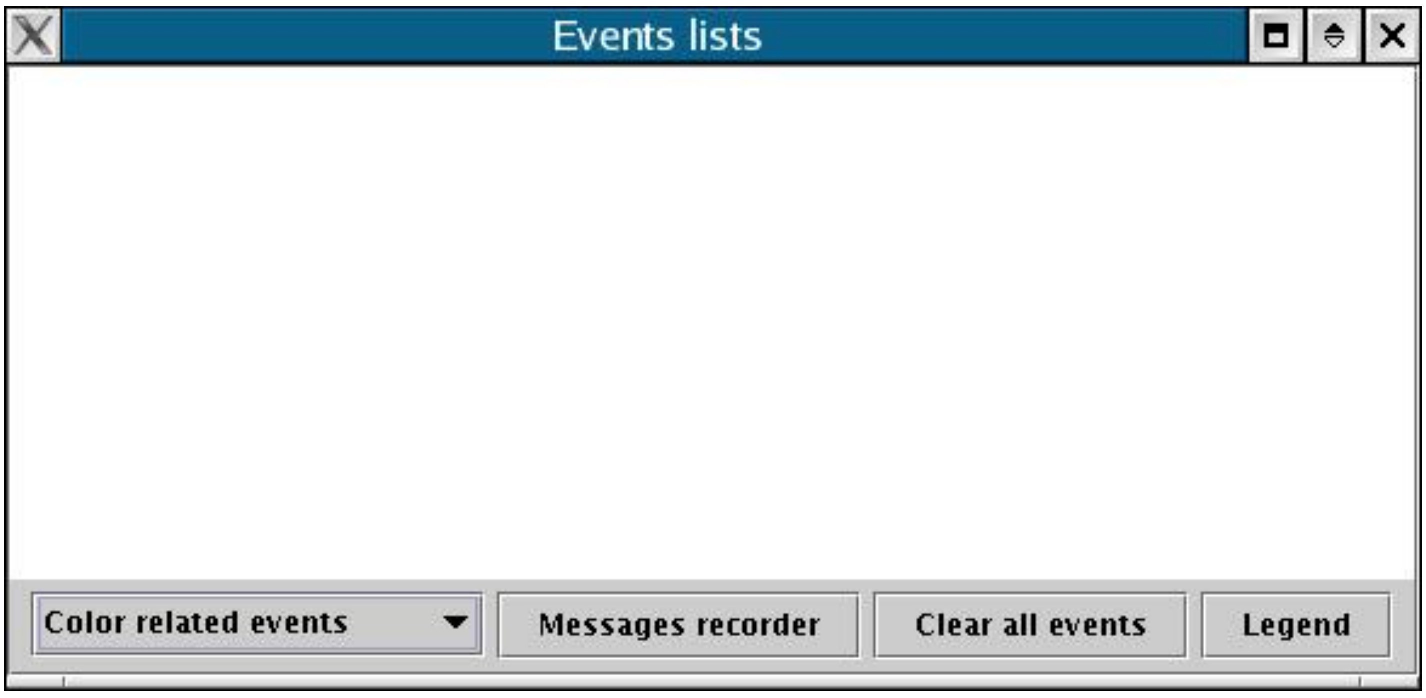
```
C:\WINNT\system32\CMD.EXE - ic2d.bat

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
_
```



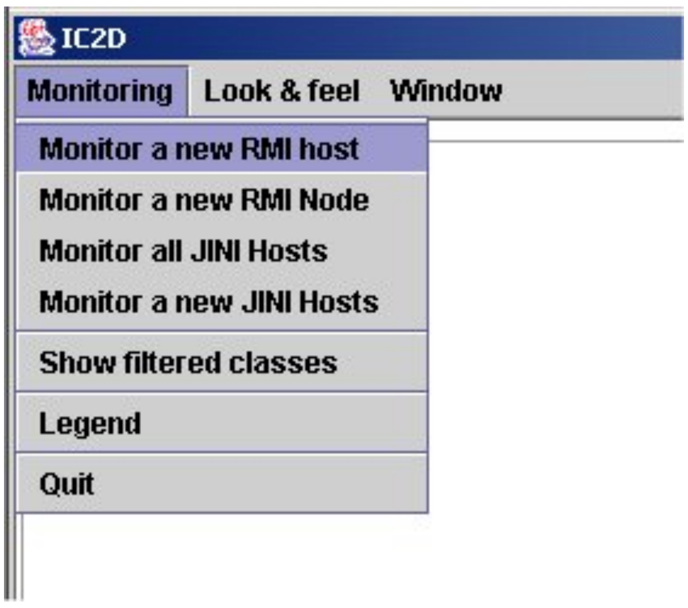
18:43:35 (AWT-EventQueue-0) --> IC2D ready !



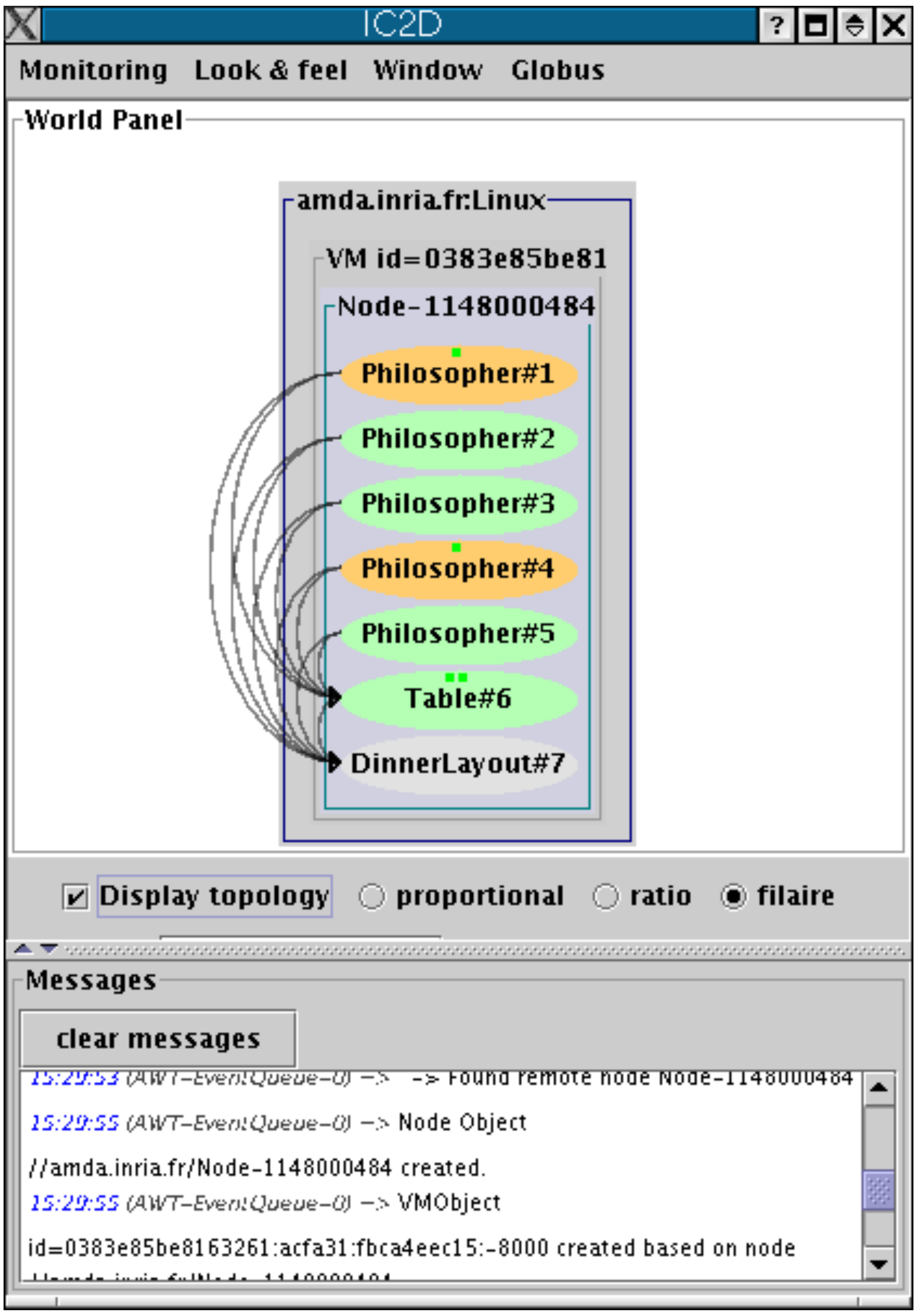
the ic2d GUI is started. It is composed of 2 panels : the main panel and the events list panel

− acquire you current machine

menu monitoring − monitor new RMI host

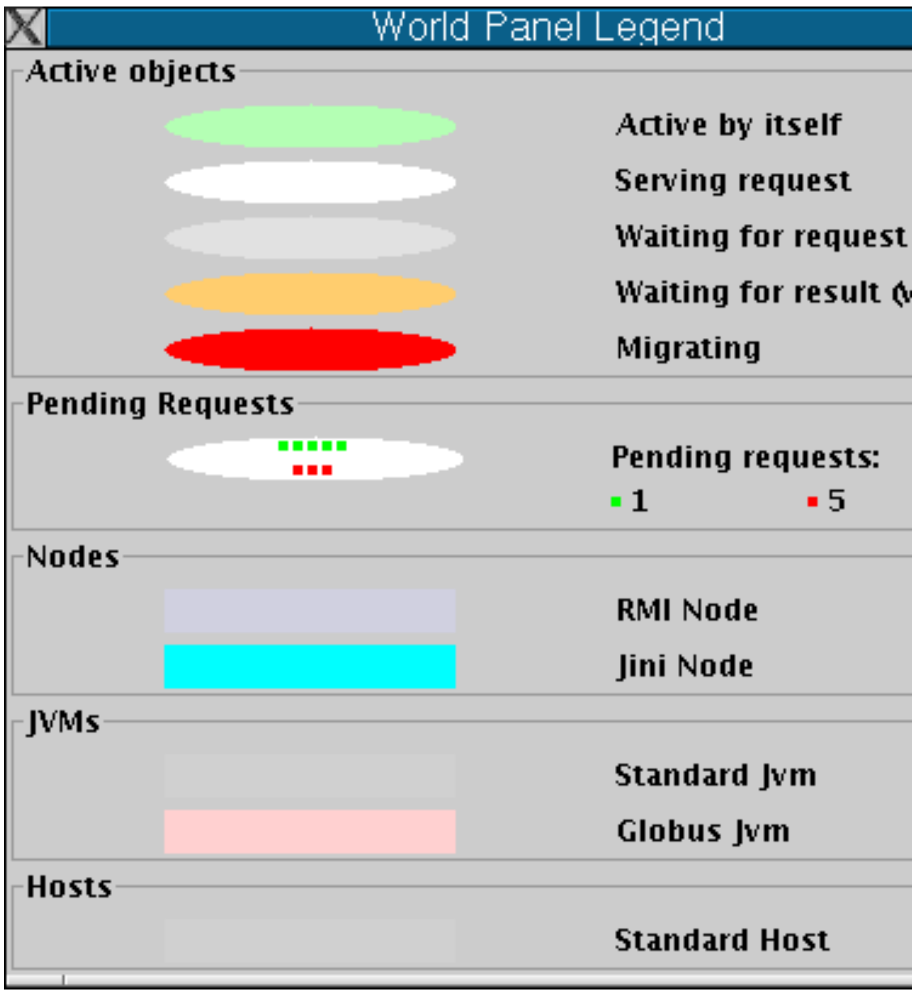


It is possible to visualize the status of each active object (processing, waiting etc...), the communications between active objects, and the topology of the system (here all active objects are in the same node) :



☑ Display topology ○ proportional ○ ratio ● filaire

```
15:29:53 (AWT-EventQueue-0) ->   -> Found remote node Node-1148000484
15:29:55 (AWT-EventQueue-0) -> Node Object
//amda.inria.fr/Node-1148000484 created.
15:29:55 (AWT-EventQueue-0) -> VMObject
id=0383e85be8163261:acfa31:fbca4eec15:-8000 created based on node
```



2.2. Parallel processing with ProActive

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

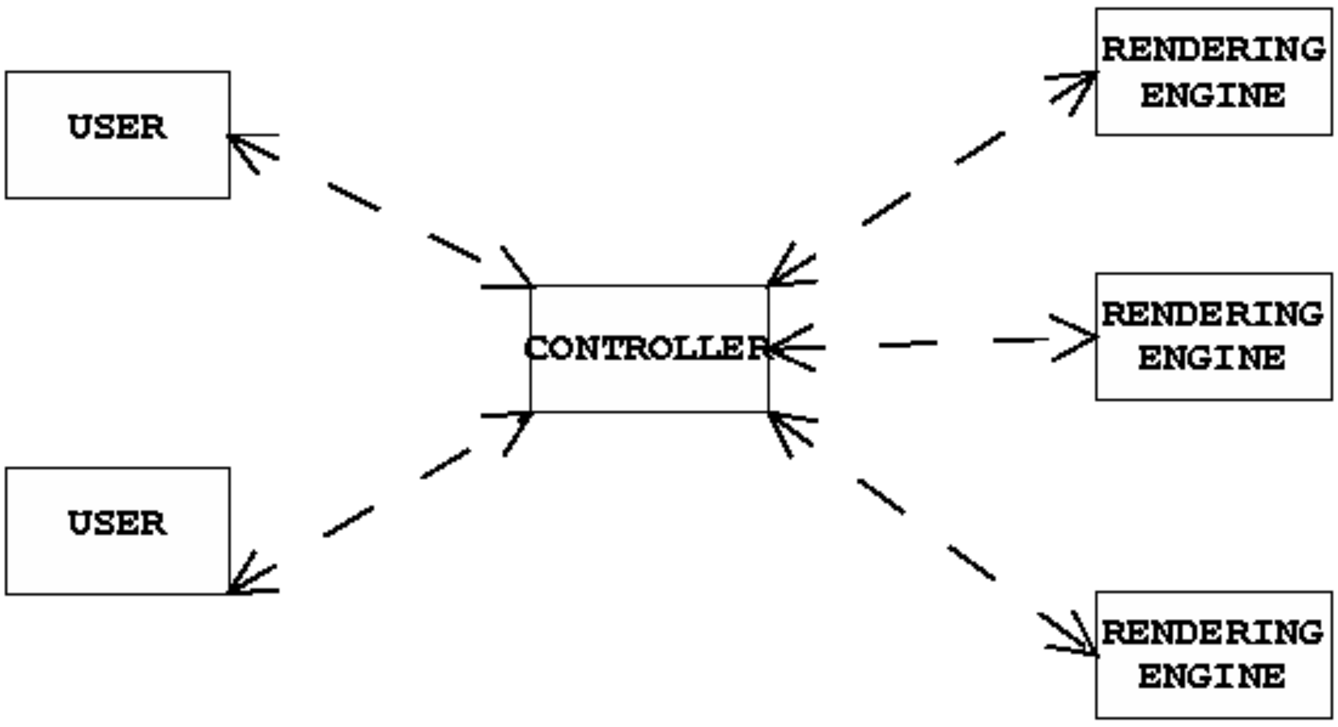
A typical application is C3D.

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.

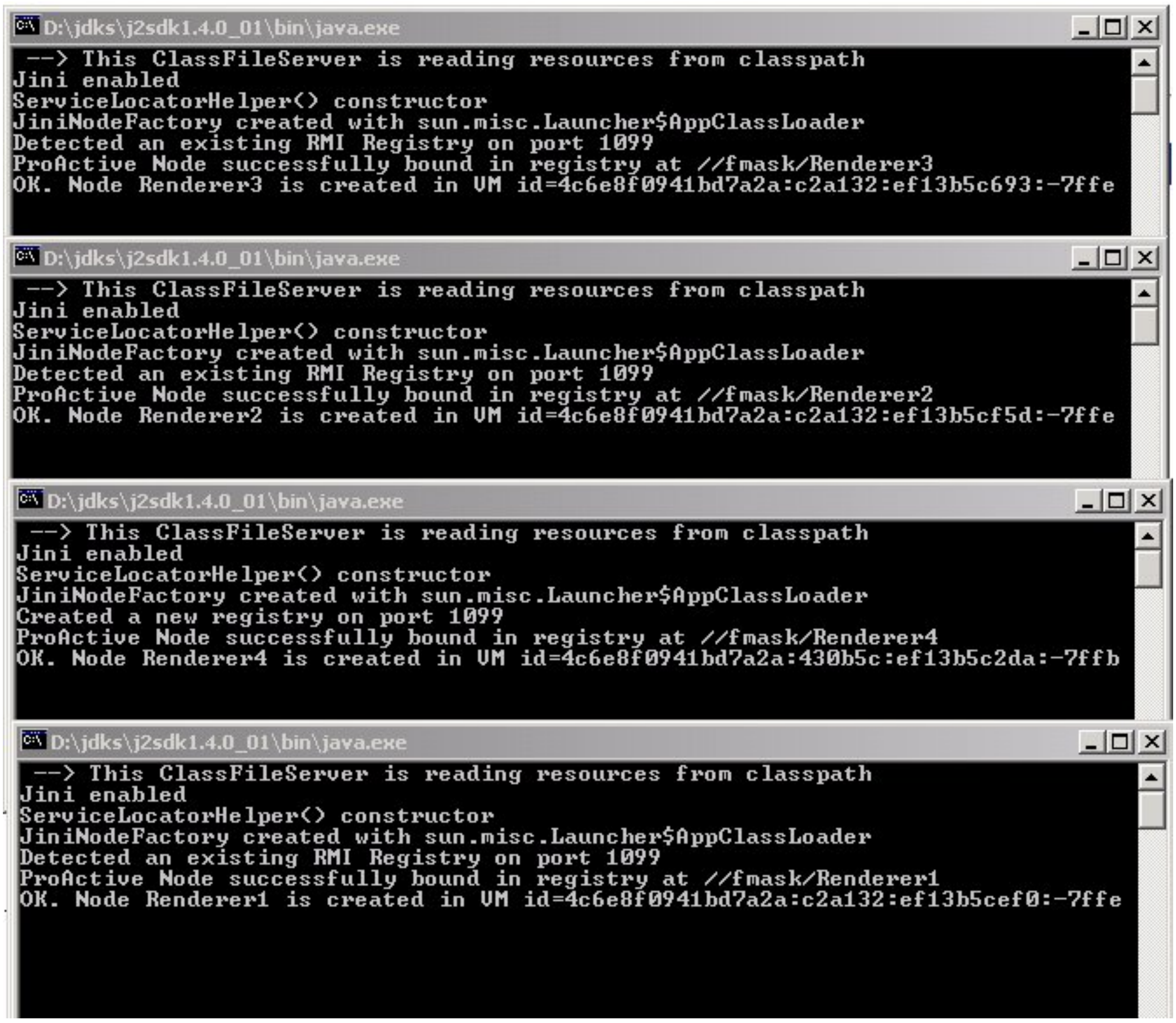


the active objects in the c3d application

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.



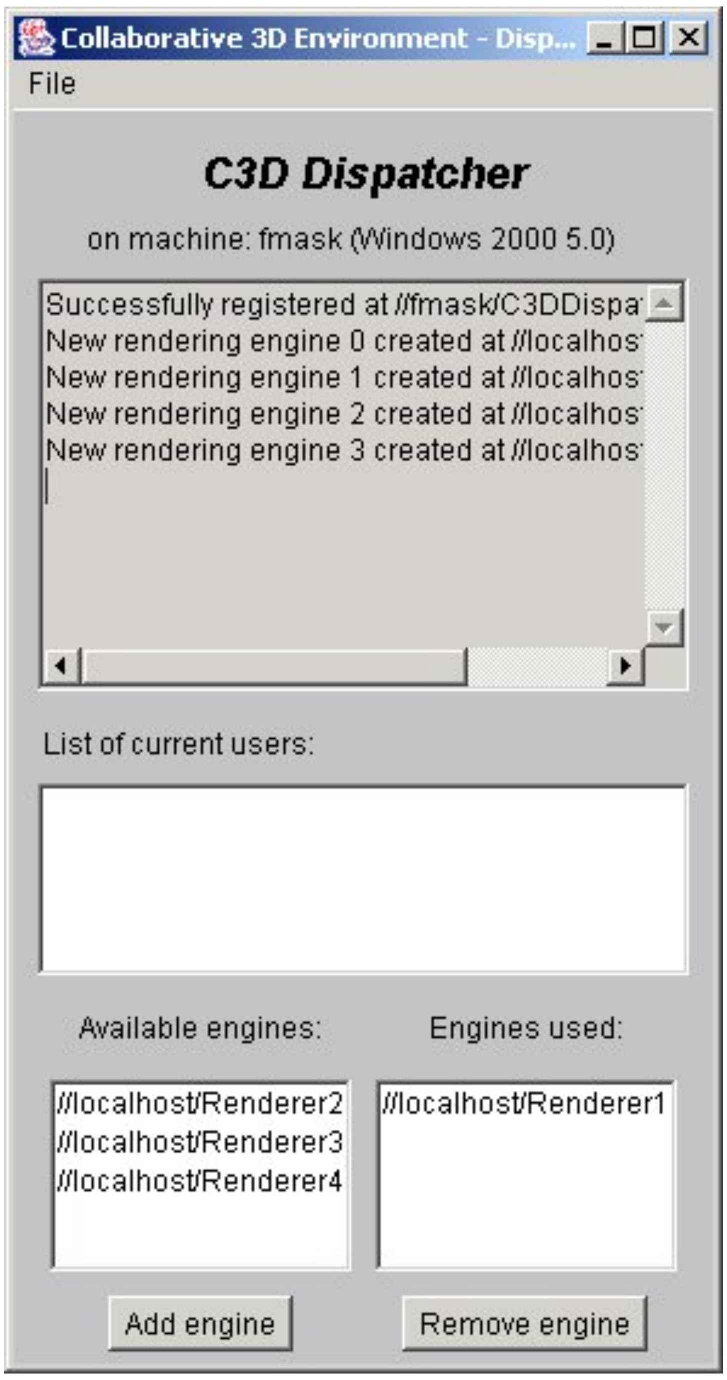
```
 --> This ClassFileServer is reading resources from classpath
Jini enabled
ServiceLocatorHelper() constructor
JiniNodeFactory created with sun.misc.Launcher$AppClassLoader
Detected an existing RMI Registry on port 1099
ProActive Node successfully bound in registry at //fmask/Renderer3
OK. Node Renderer3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

```
 --> 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 VM id=4c6e8f0941bd7a2a:c2a132:ef13b5cf5d:-7ffe
```

```
 --> 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 VM id=4c6e8f0941bd7a2a:430b5c:ef13b5c2da:-7ffb
```

```
 --> 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 VM id=4c6e8f0941bd7a2a:c2a132:ef13b5cef0:-7ffe
```

the 4 renderers are launched



C3D Dispatcher

on machine: fmask (Windows 2000 5.0)

```
Successfully registered at //fmask/C3DDispa
New rendering engine 0 created at //localhos
New rendering engine 1 created at //localhos
New rendering engine 2 created at //localhos
New rendering engine 3 created at //localhos
```

List of current users:

Available engines:	Engines used:

//localhost/Renderer2
//localhost/Renderer3
//localhost/Renderer4
	//localhost/Renderer1

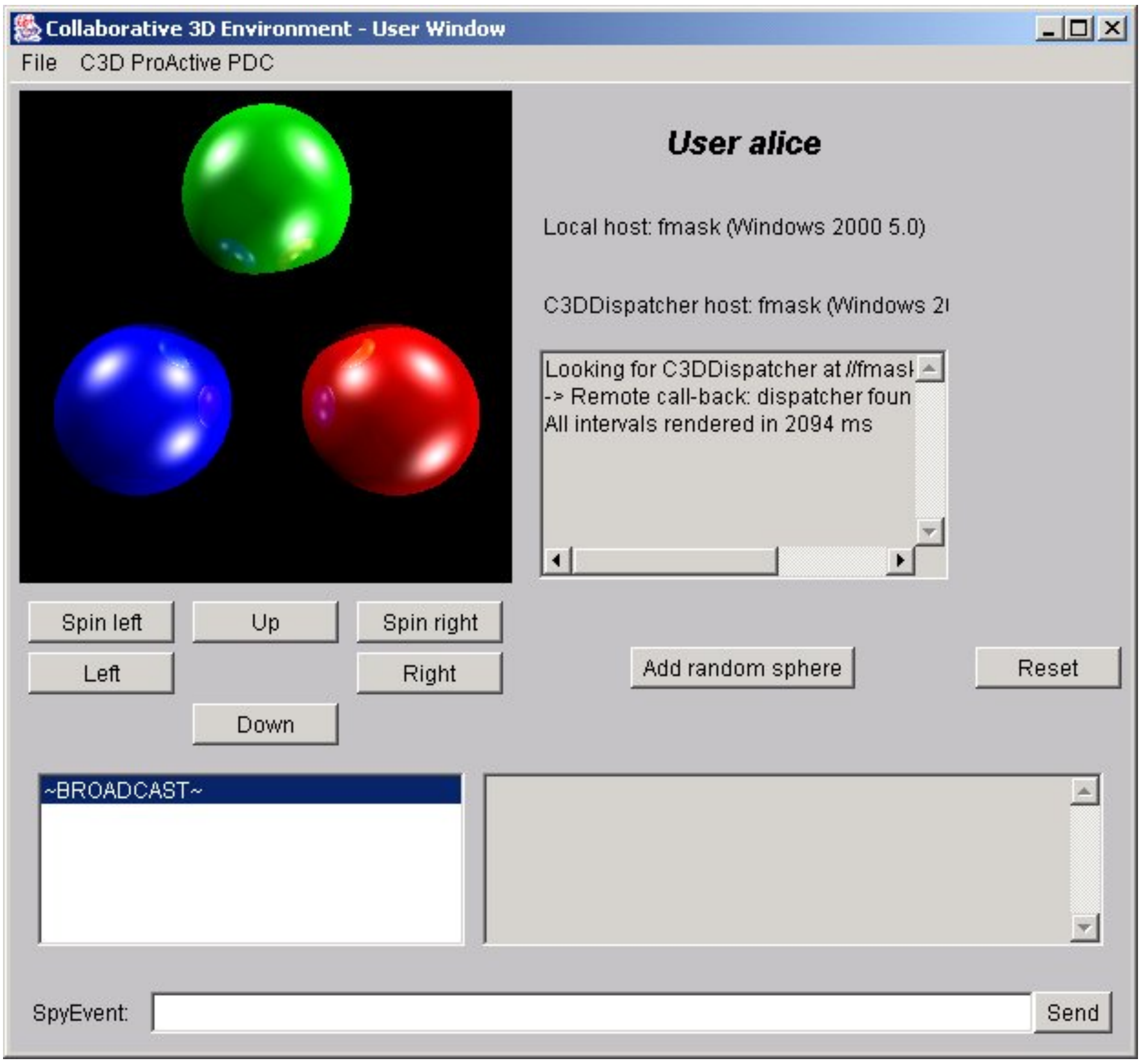
the dispatcher GUI is launched

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



User alice

Local host: fmask (Windows 2000 5.0)

C3DDispatcher host: fmask (Windows 2

```
Looking for C3DDispatcher at //fmas
-> Remote call-back: dispatcher foun
All intervals rendered in 2094 ms
```

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.

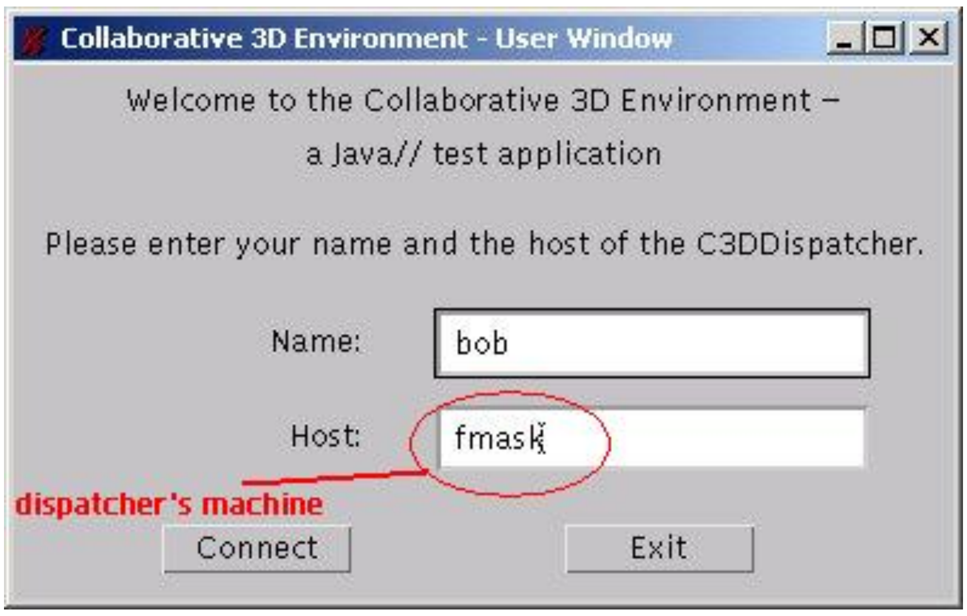
Which configuration is the fastest for the rendering?

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)



Welcome to the Collaborative 3D Environment –
a Java// test application

Please enter your name and the host of the C3DDispatcher.

Name: bob

Host: fmask

dispatcher's machine

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

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

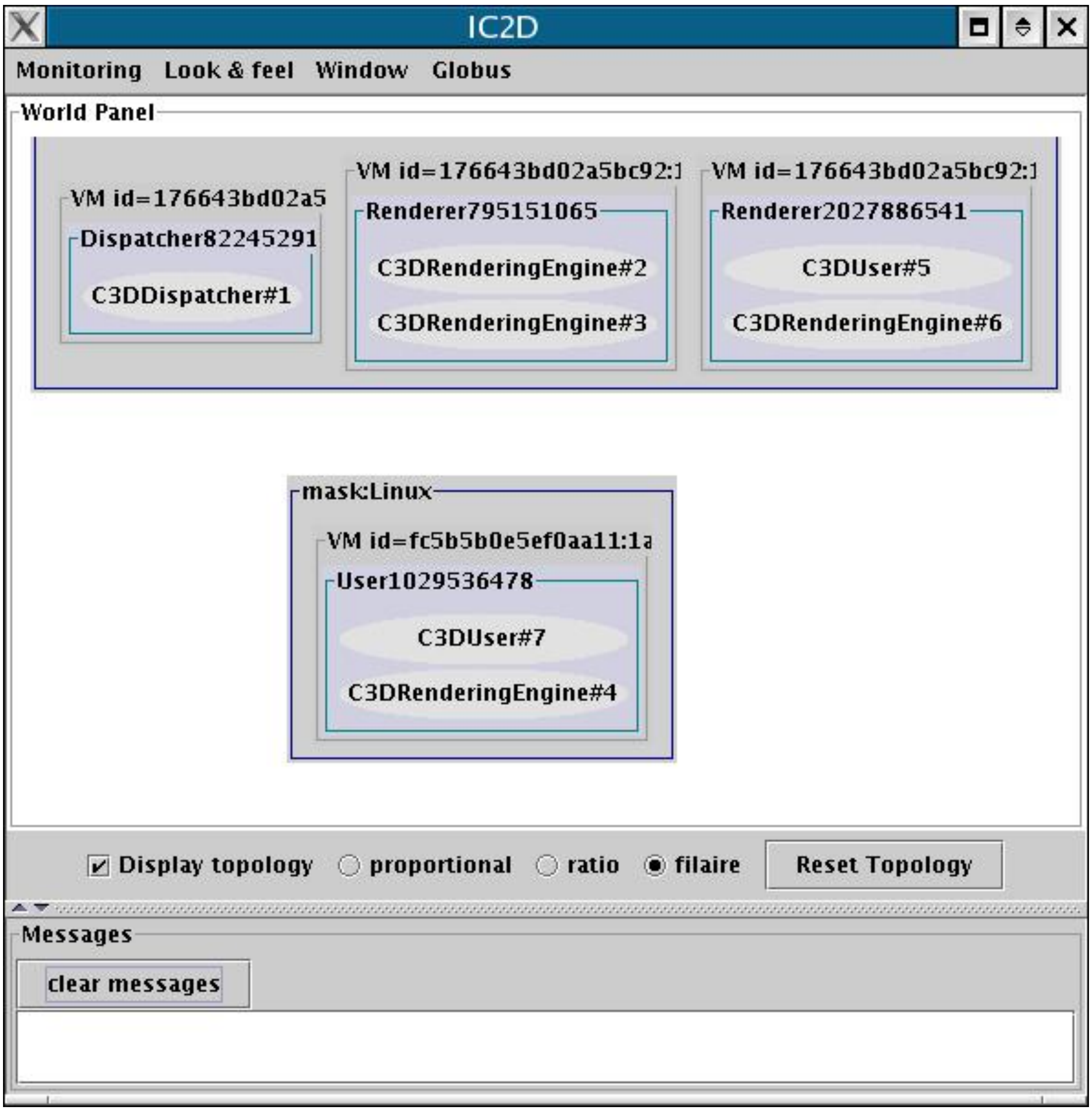
− test the collaborative behavior of the application when several users are connected.

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

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



☑ Display topology ○ proportional ○ ratio ● filaire

− 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.

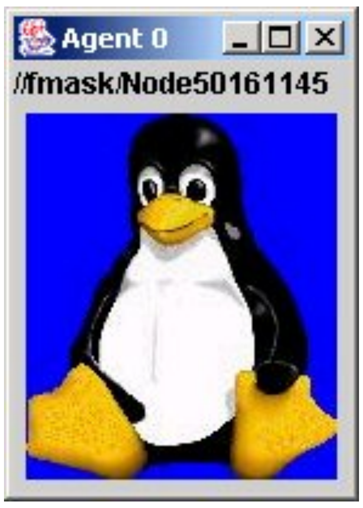
2. start IC2D to see what is going on

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.

Code examples

```
/**
 * 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);
    }
}
```

```
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");
}
}
```

```
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.getNa
                    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 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);
        }
    }
}
```

```
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 explicitely 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());
        }
    }
}
```

```
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) { // instanciation-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");
        }
    }
}
```

3. Conclusion

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

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

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

Further information can be found on the website, and suggestions are welcome.