```
    * @return what the agent has to say
    */
    public void sayHello() {
        //System.out.println("\nHELLO!");
        System.out.println("\n---------------------\nhello,from : " + getLocalHostName());
    }

    public void toto() {
    }

    /** locator method
    *
    * @return the name of the host currently containing the object
    */
    protected String getLocalHostName() {
        try {
            return InetAddress.getLocalHost().toString();
        } catch (UnknownHostException uhe) {
            uhe.printStackTrace();
            return "! localhost resolution failed";
        }
    }

    /**
     * Returns the name.
     * @return String
     */
    public String getName() {
        return name;
    }
```

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 public String getName() {

Create 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 the last parameter of `newActive` is set to `null`. In order for the client to obtain an initial referenceonto this remote object, we need to register it

ProActive PDC 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 PDC 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 o 0 ld // the saF4 1Tj 0  myates th=567
```

```
clientHost> java org.objectweb.proactive.examples.hello.HelloClient //remoteHost/Hello
```

1.2. Initialization of the activity

an internal/F4read).
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.

```
	}

	/**
	 * 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)tweb.proactive.InitActive#initASystem.out.println("I
	}

	/**
	 * @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)tweb.proactive.InitActive#initASystem.out.println("I
	}


	/**
	 * 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();
	}


}
```

Execution

Execution is similar to the previous example.
In order to see the end of the activity, you will of course need to add a call to hello.terminate() !

[redacted]

1.3.2. Required conditions

− provide a no−arg constructor

− provide an implementation fndersing ProActive'. migration mechanism.

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

A factory static method is added fndeease of creation.

Here is the code fndethis class :

```
                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 is the last one of the method moveTo. See the manual for more information.

c) the main method

The entry poln of the program is written in a separate class : MigrationTutorial.

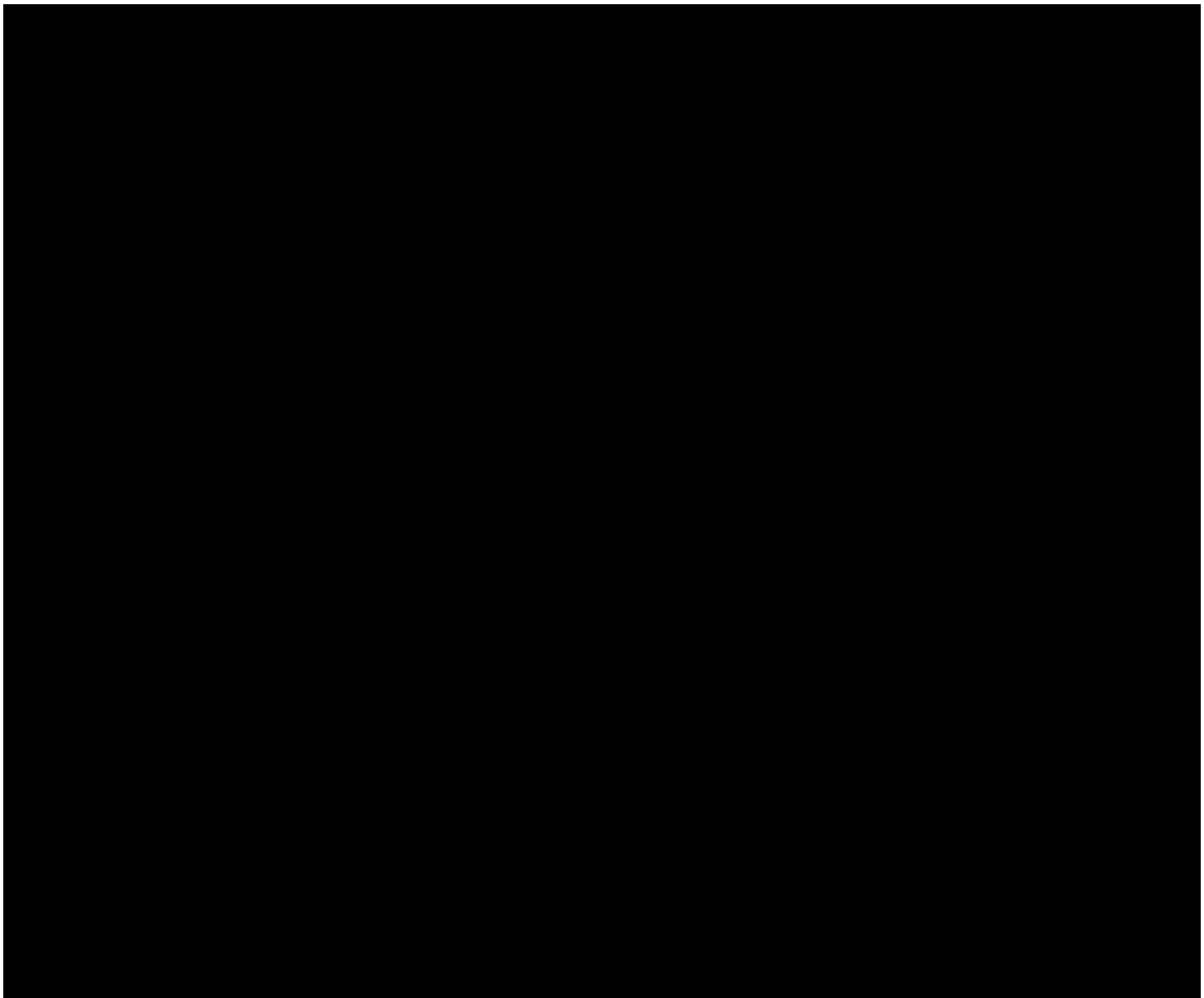
It takes as arguments the locations of the nodes the object will be migrated to.

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.

Here is the code for this main method :

```
/** entry poln 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.createMigratableH

        // check if the migratable_hello has been created
   if (migratable_hello != null) {

                // say hello
                migratable_hello.sayHello();

                // start moving the object around
                for (ln i = 0; i < args.length; i++) {
                        migratable_hello.moveTo(args[i]);
                        migratable_hello.sayHello();

                        // wait for a little while before next migration
                        try {
                                Thread.sleep(5000);
                        } catch (InterruptedException ie) {
                                System.out.println("problem while pausing :
                                ie.printStackTrace();
                                }
                        }
```

1.4. migration of graphical interfaces



HelloFrameController

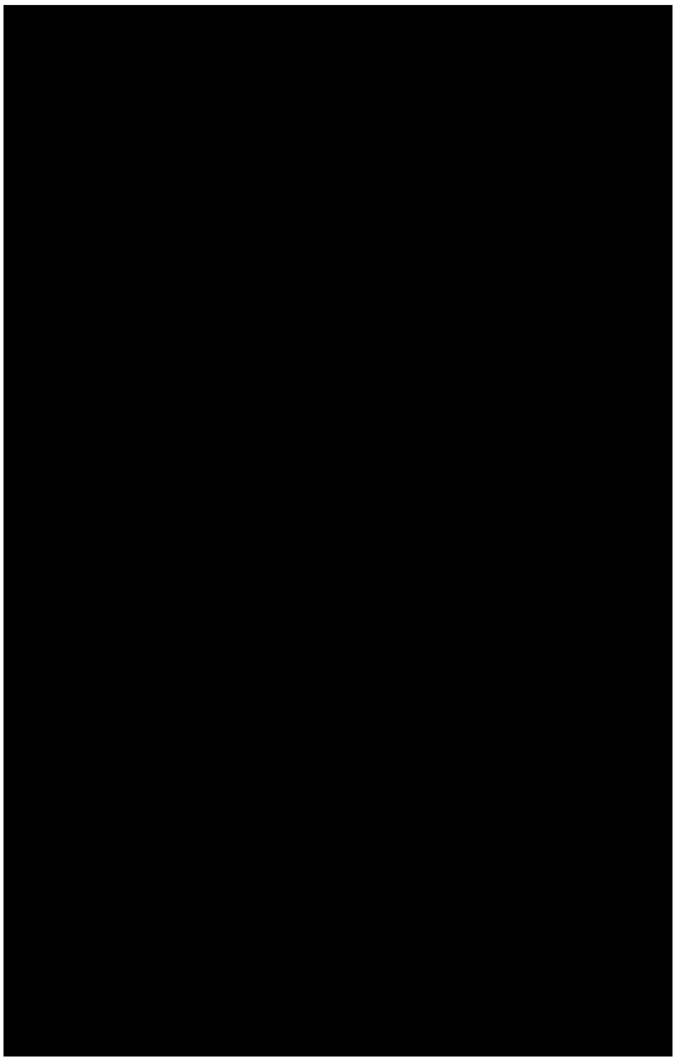
This class extends MigratableHello, and adds an activity and a migration strategy manager to the object .
It creates a graphical frame upon call of the sayHello method.

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.

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.

The activity of an active object is

```
/**
 *
 * 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 j 0 −11 Tw 0 −1eCoq   }
```



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

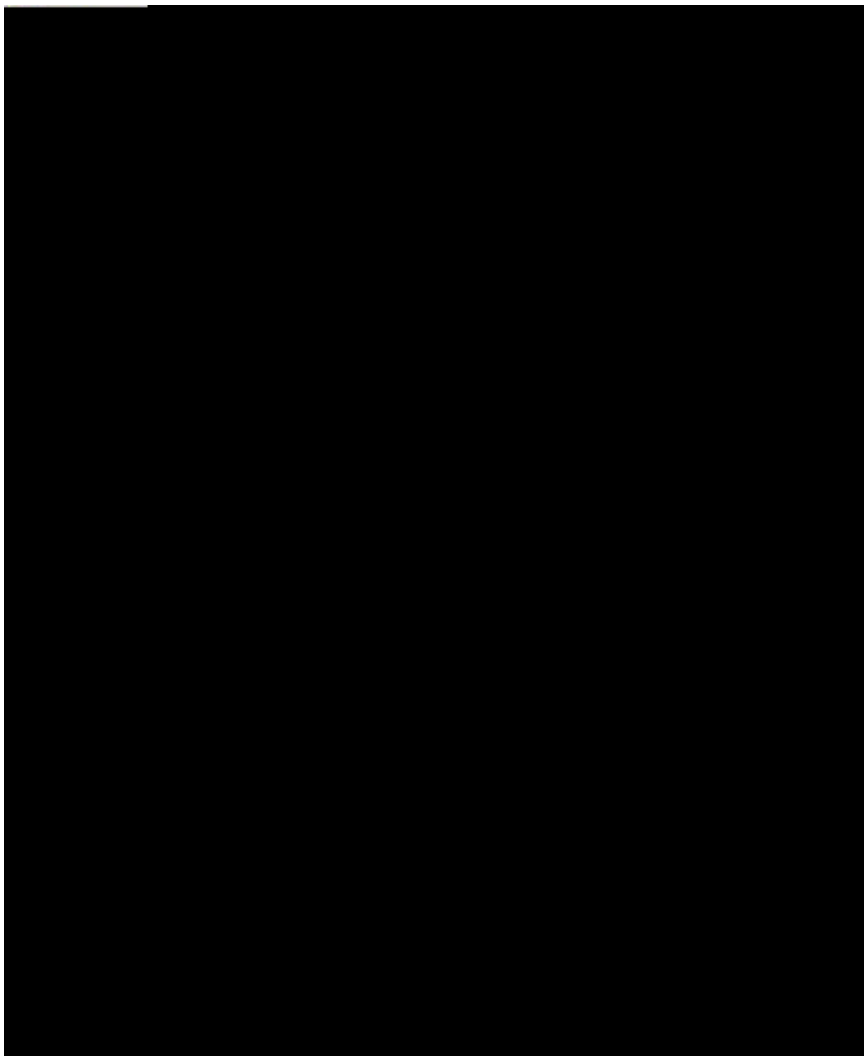
org. objecweb. po actie.examples.rReades. Reade Write.java.

org. objecweb. po actie.Service service).

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

Look at the inner class `MyRequestFilterm`

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



4. test the manual mode

Click on the philosophers' heads to switch their modes

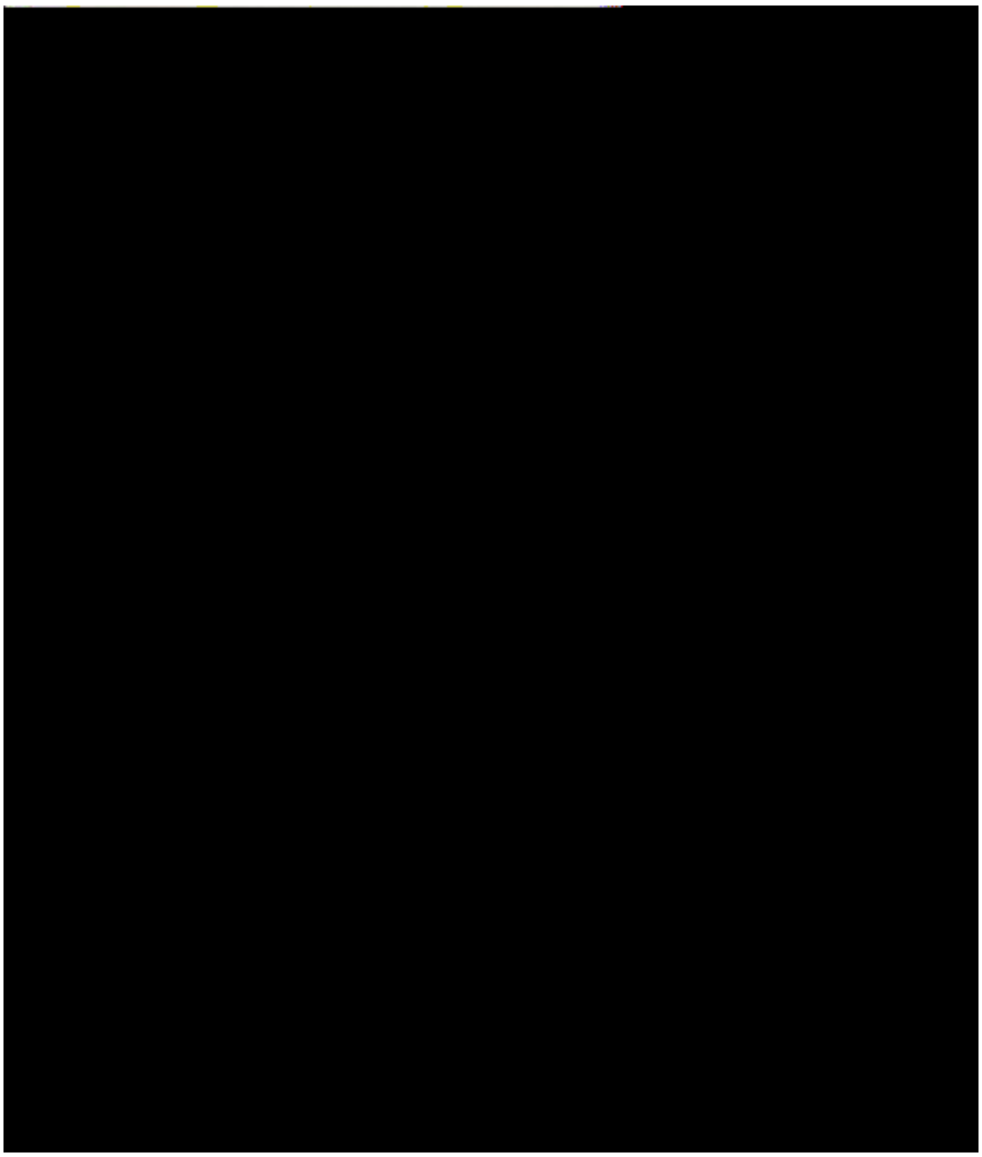
Test that there are no deadlocks!

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)



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

look at the active objects, and at the topology

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 cac0.8.sacilitated with ProActive, thanks to asynchronism method calls, as well as group communications.

C3D : a parallel, distributed and collaborative 3D renderer

C3D is a Java0.8nchmark application that measures the performance of a 3D raytracer renderer distributed over several Java0virtual machines using Java0RMI. It showcases some of the .8nefits of ProActive, nctably the ease of distributed programming, and the speedup through parallel calculation.

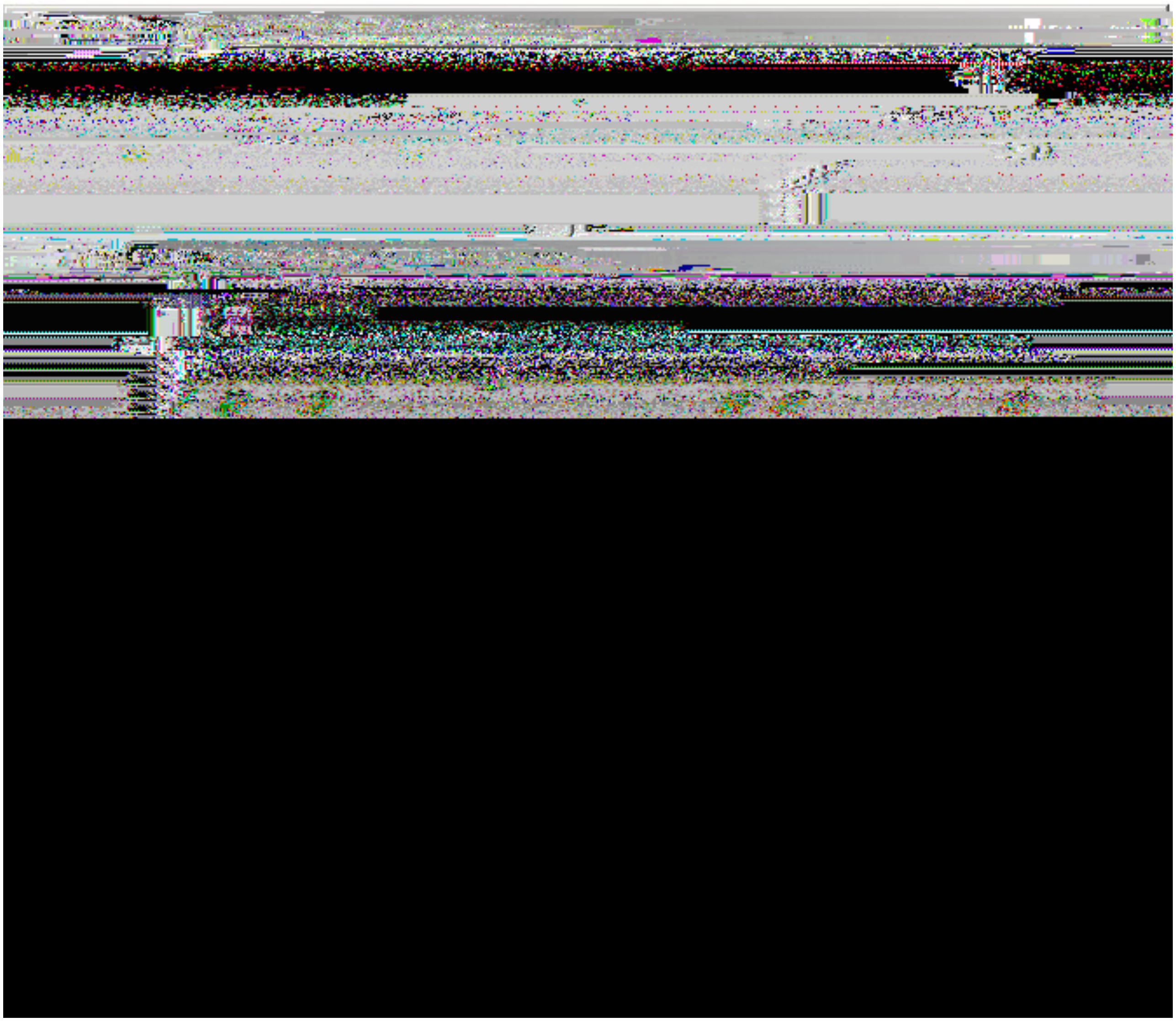
Several users cac0collaboratively view and manipulate a 3D sc8ne. The image of the sc8ne is calculated by a dynamic set of rendering engines using a raytracing algorithm, everything .8ing controlled by a c8ntral dispatcher.

the active objects in the c3d application

1. start C3D

using the script c3d_no_user

A "Dispatcher" object is launched (ie a c8ntralized server) as well as 4 "Renderer" objects, that are active objects to be used for parallel rendering.



the 4 renderers are launched

the dispatcher GUI is launched

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

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.

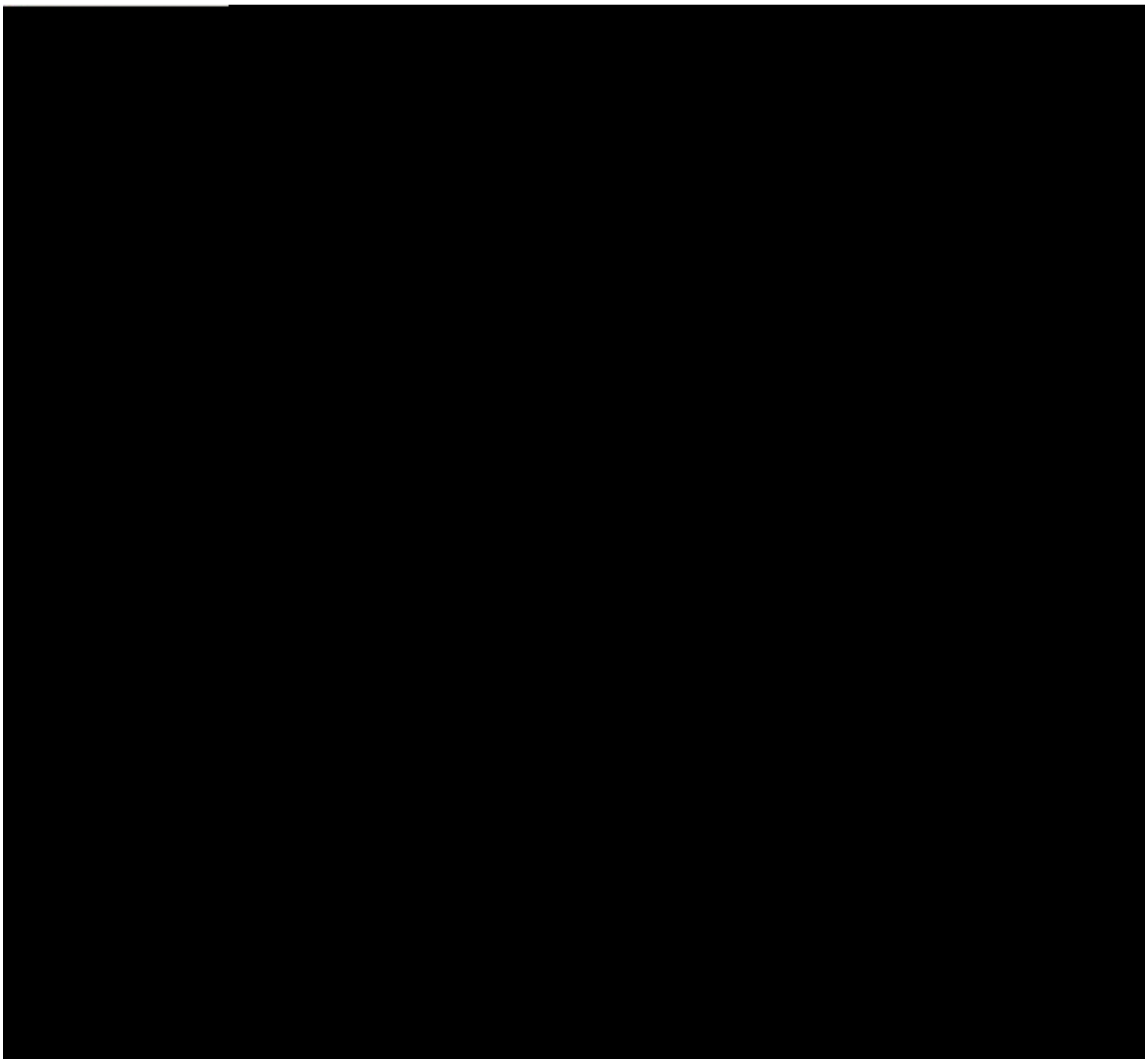
Notice that a collaborative consensus must be reached before starting some actions (or that a timeout occured).

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



− 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.

− 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 on of the renderers, and check that the node is now taking place in the computation :

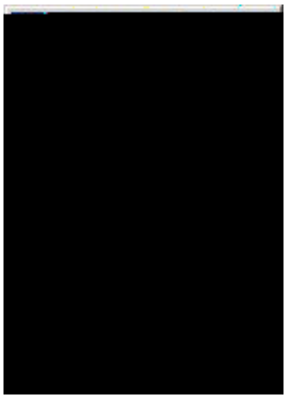
2.3. Migration of active objects

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 thee60 rse and with each other. It also features the capability to move a swing window between s86 arof.dd8w8 ihile still communQrome p



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.