DRM Programming Tutorial

Márk Jelasity

1 What is this tutorial about?

It is about writing applications which are based on the DRM API. The DRM (distributed resource machine) is an environment for special multiagent applications. A DRM is a pure fully distributed peer-to-peer network of nodes which use epidemic protocols to communicate with each other. A DRM offers functionality supporting agents and *collectives* (see Section 7).

This tutorial is useful for those who wish to use the DRM directly and not through higher level libraries, or those who use higher level libraries but want to access lower level functionality as well.

Section 3 guides the user to running the famous "Hello world!" application. This will give a basic idea of how a DRM application looks like and how does it feel to run something on a DRM. Section 4 gives the reader an overview of the functionality the applications can use. The remaining sections discuss examples in detail. The example in Section 7 is relatively more complex, in fact it illustrates a potentially promising approach to writing real and complex applications.

2 What is this tutorial not about?

This tutorial is a practical guide to the API documentation. That is, it is *not* a class documentation *nor* an introduction to the low level functioning of the DRM. It does not contain other specifications either, like details on configuration, tools, security, and technical details on starting experiments. These things can be accessed in the class documentation, and in the documents linked to the overview page. of the class documentation in directory `doc/api`[1]. There will be many references to that documentation throughout this document. The html version of this document contains links to it as well.

3 Running the first application

3.1 Setting up a local DRM

To run an application it is necessary to have a running DRM (distributed resource machine). A DRM is made up of a set of nodes, which are connected to each other.

[1] From now on we will assume that the reader installed the DRM properly.

The DRM does not rely on a server, the nodes communicate with each other directly.

The idea is that there is a "big DRM" out there on the Internet which unites all participating nodes. When developing applications, it is a good idea however to set up a local DRM and run the applications there. The reason is that if the applications have bugs for example, it is much harder to take action once the application is out on the big DRM.

Accordingly, we will set up a local DRM for trying out the example applications explained in this document. In the simplest case one can start a DRM which consists of only one node. This is done by running the application `drmnode`. (In fact this starts the class `drm.server.TestGUINode` with proper security and classpath settings.)

It sounds simple enough. However, already at this point we have to make a comment. To start a DRM of one node it is necessary to make sure that no other nodes connect to our single node. If our computer is not on the network, it will not happen. However, if our computer is connected to the Internet and from the same computer a node was part of the big DRM before, there is some chance that the big DRM remembers us and connects our node again. To make absolutely sure that we are alone, we can use the property "group" which identifies the DRM. The group of the big DRM is called "default". Any different name will prevent connection. The group name can be configured, see the configuration guide in the class documentation. The command line approach would look like this:

```
drmnode group=myUniqueGroupName
```

Building of DRMs of more than one node is described in Section 5.1.

3.2 Running an application

The technical specification of starting experiments (jobs) is described in the experiment starting specification in the class documentation. From now on we will assume that the experiment is in a jar file and not in a directory, noting that a directory can be more flexible when developing an application, but the agents started from a directory and not a jar are not realy mobile (see the experiment starting specification). To recompile the examples you have to include the jar `drm.jar` in the classpath. This file can be found in the DRM installation directory, under the directory `lib/`.

To run the "Hello world!" application it is enough to have a DRM of one node. Start the node as described above. Select the File/Run menu. In this dialog you have to type `examples/1.jar` and hit enter. In fact you have to give an absolute pathname to `1.jar` or a relative one from the directory the node was started in.

You should see an agent appearing in the agent list and the text "Hello world!" on the screen. After a couple of seconds the agent disappears. Note that our agent writes to the standard output so you can see its message only if you start the application from a console (shell). (For example, in Windows starting drmnode by clicking its icon will start the node, you can start the experiment too but you will not see the message.)

3.3 Security

By default applications have sandbox rights, they are not allowed to write to the disk or to initiate network connections for example. Our simple example in `1.jar` did not need this but we normally want our own experiments to be allowed to write on our own local disk. The solution is that it is possible to give an experiment application rights in the node it was started in. To achieve this, the jar file which contains the experiment must be under the directory from which the node was started at the time of launching. Note that this does not imply that this experiment will have application rights anywhere else.

3.4 The application from the inside

The technical specification of classes to be included in an experiment can be found in the experiment starting specification in the class documentation. Our job contains two classes. The contents of the jar that contains the job looks like this:

```
hwjob/HelloWorld.class
Launch.class
```

Here the class `Launch` is the standard class that launches the job. It functions like the `main` method in C or Java applications. The launcher normally launches agents. The other class is the agent that actually performs the job.

The source code of `Launch` is in Section A. It implements the standard constructor and the Runnable interface, these are required in each Launcher class. The run method constructs an agent and uses the `launch` method of the node to launch it. Launch type `''DIRECT''` means that the agent is launched to the node address given in the third parameter which is null in our case which means the agent is put to the node on which `launch` was called. For more information on launching you can refer to the class documentation of `drm.core.Node` and `drm.agentbase.IBase` which is an interface implemented by `Node`.

The source code of the agent is in Section B. The source code speaks for itself, especially combined with the class documentation of `drm.agents.Agent`. The only requirement is that an agent must implement the interface `drm.agentbase.IAgent`. The class `drm.agents.Agent` is a convenience class that makes writing agents much easier as most methods have a default implementation. In our case we implemented only the `run` method, which simply writes the message on the standard output. There is one important thing to note: to stop running, and agent must explicitly call `suicide()` (line 21) which will result in removing the agent from the environment. Without calling this method, the agent stays alive, although it becomes passive. Staying alive means that it can still receive and handle messages for example (see Section 6).

4 The API: an overview

Now that we have seen an example that actually works, it will be easier to give a general overview of the API that is available to the developers. Even though the basic

design concepts of a job are not yet clear, this overview will help the reader navigate the class documentation and understand the examples of the remaining sections better. The following sections describe the different parts of the API w.r.t. functionality and users.

4.1 The interface `drm.agentbase.IBase`

4.1.1 Functionality

- Agent administration, like adding, deleting, dispatching agents, etc.
- Information about the state of the node.
- Message sending.

4.1.2 Users

- The launcher class, since the `Node` object passed to its constructor implements this interface.
- The agents, after the agentbase calls their method `setBase`. This always happens when the agent is put into a base, before its `run` method is started.

4.2 The interface `drm.core.IDRM`

4.2.1 Functionality

Providing information about other nodes that are part of the same DRM.

4.2.2 Users

The agents, after the agentbase calls their method `setBase`. This always happens when the agent is put into a base, before its `run` method is started. Extending classes of `Agent` can reach it through `getDRM`.

4.3 Public and protected interface of `drm.agents.Agent`

4.3.1 Functionality

- Signs the state of the agent via the flag `shouldLive` that must be used by extending classes to properly shut down any threads when this flag becomes false.
- It provides a couple of convenience functions that simplify sending messages via simpler interfaces.
- It defines methods for debugging suitable for redefinition in extending classes.
- Access to the hosting environment is provided through the field `base` and through the function `getDRM()`.

4.3.2 Users

Agents that extend `Agent`.

4.4 Public and protected interface of `drm.agents.ContributorAgent`

4.4.1 Functionality

- Extends `Agent` by adding the functionality of being able to participate in a *collective*. In particular, it implements the interfaces `drm.core.Observer` and `drm.core.Contributor`. Section 7 discusses collectives in more detail. The collective can be accessed through the field `collective`.
- Implements the concept of *root island* in jobs. That is, jobs will have a root island which will serve as a bootstrap device for hooking up new agents to the job collective. This concept will be illustrated in Section 7.

4.4.2 Users

Agents that extend `ContributorAgent`.

5 Mobility

We will illustrate mobility through an application which consists of one agent which jumps between different nodes while counting the number of jumps.

5.1 Setting up the DRM

In the case of the first example it was sufficient to have a DRM of one node. To illustrate jumping, we need a DRM of at least two nodes. This can be achieved quite easily: you have to start two nodes and you have to connect them. Starting two nodes involves starting the application `drmnode` two times. This results in two independent (i.e. not connected) DRMs each consisting of one node.

Connecting the nodes can be done using the dialog "edit/add to node list". Open this dialog in the node started as *second* and in the simplest case type `localhost` (or the name of your computer). If you started the nodes from a console, you should see log messages indicating that the two nodes started to communicate. An alternative way is to start the second node with the command line parameter `node=localhost`.

There is an important thing to keep in mind however. If you installed the DRM software properly, you have a node which is started up automatically and is running in the background, most likely as part of the big DRM. In that case if you simply type `localhost` in the dialog window, this node will be added which we do not want now as we are building a local DRM. You can even get an error message if the node you started is in a different group than the default node in the background.

To solve this problem we need some insight into how port assignments work. The default port is 10101, and if this is taken, the next free port is used which is 10102,

etc. However the node also remembers which port was configured at starting time thus it is possible that you have a node which uses 10102 but knows that it was configured to use 10101. Simply typing `localhost` means you have chosen the port which was originally configured. This is why you had to use the *second* node to connect the two nodes on the same machine, because of course it does not make sense to add yourself to the node list. This feature makes more sense when you connect nodes on different computers.

To control the ports explicitly you can specify a port at startup time It is done as follows:

```
drmnode group=myUniqueGroupName port=12345
```

The default port is 10101, so you might want to set a different one to avoid unwanted connections. When starting the second node, you must use exactly the same command line. The node will detect that the given port is not free and will automatically use the next one, 12346 in this case. Typing `localhost` in this second node will asume the port 12345 as described above.

You can also specify the port explicitly in the dialog by using the syntax as in `localhost:10102` for example.

Using the procedure described above you can build a DRM of any size. The nodes do not have to be on the same computer. As a node it is possible to use not only the node started by `drmnode` by default, you can apply the `--nogui` option which results in starting a node without a window. (This in fact means running the class `drm.server.NakedNode`.) This application understands the same command line arguments and configuration files, only it does not have a graphical interface (in fact any interface...) so it is suitable to be run in the background. Be careful because `drmnode --nogui` tries to connect to the big DRM by default using default nodes. To prevent this, you can use a different group name (see Section 3.1).

5.2 The application from the inside

The jumper application is in `examples/2.jar`. It is advisable to set up a local DRM of two nodes with a graphical interface started from to different consoles, and start this application on any of the nodes. The agent starts jumping immediately and writes messages to the consoles as well.

The launcher class (Section C) is almost identical to that of the first example. The code of the agent is in Section D. The agent uses the `IDRM` interface (Section 4.2) to get information about the other nodes in the DRM. Its actions are determined by this information, and a serializable field that counts the jumps (line 12). The value of this field is preserved according the java object serialization principles when the agent is traveling to another node.

6 Communication

This little application involves an agent which itself launches another agent, sends it a message and prints the answer. It illustrates message sending and the fact that

agents can create and launch agents as well through the `IBase` interface they access (Section 4.1).

The application works on a single node or on a DRM with many nodes as well. Start the file `examples/3.jar` from a node.

The launcher class has the same simple structure again (Section E). The agent is in Section F. The agent started first has the full name `Talker.test-<time>.1` as determined by the launcher and the agent. The agent started by the first agent has the same name except that it ends in 2 and not 1. The test in line 20 checks if the running agent is the first or the second. The second agent is passive, i.e. its run function exits immediately. It only waits for messages. The first agent attempts to launch the second agent.

Observe the usage of the interface `drm.agentbase.IRequest`. Every functionality that is related to communication is asynchronous. This is because the system is supposed to work under very poor conditions as well. The methods that implement these functionalities return an `IRequest` object. If you want to make sure an operation was successful, you have to wait until it is finished and check its status, like this agent does.

Asynchronous operations often return some information too which is always documented in the API documentation of the corresponding method. In the case of launch, you can ask for the address of the launched agent if the launch was successful and the launch type is `''RANDOM''` (line 31). In the case of message sending, you can ask for the reply (line 39).

The agent overrides the default implementation of `handleMessage`. This method is called by the node if the agent receives a message. The task of this method is to handle the message, i.e. do something and/or reply. It does not matter if the agent is passive or active (i.e. its run method is still running) when it receives a message.

Line 48 has to be always the first line of every handleMessage method that overrides the same method of the super class. The reason is that the super class may also want to handle some messages which might be of crucial importance to the functioning of the DRM. It is also important to return false if the message could not be handled to allow correct message propagation and error handling. This implementation handles messages of type “test” and always replies with the string “Fine thanks.” independently of the content of the message.

7 Collectives and more

After getting familiar with the agent functionality like mobility and messages, we now discuss the powerful concept of *collectives*, and an application template that fits well into this framework. The application will be an optimizer which searches for the maximum of a one dimensional real valued function. Note that this is an illustration only, what matters is the way the application is put together.

7.1 Collectives

A collective is a set of entities (agents, humans etc) who work together to achieve a goal. The main concept of a collective is the *contribution repository*. Members of a collective who are *observers* can read this repository and *contributors* can write it. The philosophy behind this model is that the goal of the collective is solved by the participants via generating contributions using the contributions of the others. These new contributions then help the others to generate contributions, and so on. During this process the state of the collective should converge to the goal.

The collective contains a command database as well, which can be written by members who are *controllers*. The members of the collective should perform the commands in the database.

A job can be a collective. The class `drm.agents.ContributorAgent` extends `Agent` with the functionality of participating in a collective. That is, if an application wants to be a collective, it can use agents that extend `ContributorAgent`. The interface is very simple. To fulfill the role of a contributor, the agent has to override the method `getContribution` which must return the most recent contribution of the agent when called by the collective. To query the contribution repository, the agent can use the public interface of the field `collective` which is of type `drm.core.Collective`.

To actually build the collective, `ContributorAgent` uses a default and reliable agent, the *root* agent. The role of this agent is that when new agents have to be added to the collective, the information which is necessary for this operation is fetched from this agent. (Note that other, more robust implementations of building a collective are possible as well, and can be expected in the future.) The root has no further role, only if it is given a further role by the application developer. In our example the root saves the contributions of the rest of the agents to disk during the running of the job.

7.2 The Files

The listing of `examples/4.jar` looks like this:

```
Launcher.class
config.properties
drmhc/HillClimber.class
drmhc/Algorithm.class
```

The difference from the previous examples is the presence of a configuration file in the root directory, and a helper class in package `drmhc`. Classes of the experiment can load the configuration file as a system resource, as we will see later, which offers a possibility to configure experiments.

7.3 The Launcher

This application has a more sophisticated launcher (Section G). It is more sophisticated because (apart from defining a separate method for launching) it does three things more:

- Error checking: using the request handler it makes the asynchronous launch operation look synchronous to the run method. In general it is not a good practice because the launcher might hang but it illustrates how to use request handlers.
- Root agent: It launches a root agent separately and then passes its address to all the new agents. This is the proper way to start up a collective using a root agent.
- Configuration file: It loads the configuration file (line 66) from the classpath as a system resource (see Section J).

7.4 The Agent

The run method of the agent performs a stochastic hillclimber search on the function `drmhc.Algorithm.eval()` (see Section I). `drmhc.Algorithm.mutate()` is used as a search operator. The agent is mobile, i.e. it can be relocated while running, because all serializable fields of the agent will keep their values. The algorithm can be configured using the configuration file (line 49) (see Section J). When finished, the agent exits, except if it is the root which is supposed to collect results (line 69) so it stays alive going passive.

In method `getContribution` the agent returns the best known solution. This is the value that is going to be written to the contribution repository time to time. Note that the class type of the contribution (in other words the data model of the application) is fully application dependent.

The agent could check the field `collective` for contributions of others. However, there is another possibility. The collective calls `collectiveUpdated` when new information arrives (line 92). Our agent uses this notification to exploit the new contributions. This is the point when the (hopefully) positive effect of learning from others manifests itself.

The root agent tries to save this information as well. The idea is that the root agent is running on the computer of the experimenter so it should be possible. Agents that run on remote nodes which are normally configured to support the sandbox security model will not be able to write anything to the disk. The filename used to save the contributions is configurable, see Section J.

Note that this is a toy example only to illustrate the basic idea. More clever launching and logging could have been implemented easily as well, our intention was to keep things as simple as possible.

A Launch

```
1  import drm.core.Node;
2  import hwjob.HelloWorld;
3
4  public class Launch implements Runnable {
5
6  private final Node node;
7
8  public Launch( Node node ) {
9
10    this.node = node;
11 }
12
13 public void run() {
14
15    node.launch(
16       "DIRECT",
17       new HelloWorld("test+"+System.currentTimeMillis(),"1"),
18       null );
19 }
20 }
21
22
```

B hwjob.HelloWorld

```
1  package hwjob;
2
3  import drm.agents.Agent;
4
5  public class HelloWorld extends Agent {
6
7  /** calls super constructor */
8  public HelloWorld( String job, String name ) {
9
10    super( "Helloworld", job, name );
11 }
12
13 /** prints "Hello World" and exits after 5s waiting */
14 public void run() {
15
16    System.out.println("Hello world!");
17
18    try { Thread.currentThread().sleep(5000); }
19    catch( Exception e ) {}
20
21    suicide();
22 }
23
24 }
```

C Launch

```
import drm.core.Node;
import jumperjob.Jumper;

public class Launch implements Runnable {

private final Node node;

public Launch( Node node ) {

   this.node = node;
}

public void run() {

   node.launch(
      "DIRECT",
      new Jumper("test+"+System.currentTimeMillis(),"1"),
      null );
}
}


```

D jumperjob.Jumper

```
1  package jumperjob;
2
3  import drm.agents.Agent;
4  import drm.core.*;
5
6  public class Jumper extends Agent {
7
8  /**
9  * jump counter. Its value is serialized so it is
10 * preserved while traveling to other nodes.
11 */
12 private int jumps = 0;
13
14 /** calls super constructor */
15 public Jumper( String job, String name ) {
16
17    super( "Jumper", job, name );
18 }
19
20 /**
21 * Jumps to another random node 3 times. The witing periods are
22 * not necessary, they are included only to slow it down so it can be
23 * followed by a human. It is supposed to be an illustration...
24 */
25 public void run() {
26
27    ContributionBox cb = getDRM().getNewestContribution();
28
29    if( cb == null )
30    {
31       System.err.println("No nodes to jump to");
32       try { Thread.currentThread().sleep(1000); }
33       catch( Exception e ) {}
34       suicide();
35    }
36    else if( jumps++ < 3 )
37    {
38       try { Thread.currentThread().sleep(1000); }
39       catch( Exception e ) {}
40       System.err.println("Jumping to "+cb.contributor);
41       base.dispatchAgent(name,cb.contributor);
42    }
43    else
44    {
```

```
45          System.err.println("Got tired of jumping around...");
46          try { Thread.currentThread().sleep(1000); }
47          catch( Exception e ) {}
48          suicide();
49       }
50    }
51
52    }
```

E Launch

```
1  import drm.core.Node;
2  import talkjob.Talker;
3
4  public class Launch implements Runnable {
5
6  private final Node node;
7
8  public Launch( Node node ) {
9
10    this.node = node;
11 }
12
13 public void run() {
14
15    node.launch(
16       "DIRECT",
17       new Talker("test+"+System.currentTimeMillis(),"1"),
18       null );
19 }
20 }
21
22
```

F talkjob.Talker

```
1  package talkjob;
2
3  import drm.agents.Agent;
4  import drm.agentbase.*;
5
6  public class Talker extends Agent {
7
8  /** calls super constructor */
9  public Talker( String job, String name ) {
10
11    super( "Talker", job, name );
12 }
13
14 /**
15 * Launches another agent, sends it a message and prints reply.
16 * Does not suicide to allow some manual testing afterwards.
17 */
18 public void run() {
19
20    if( name.endsWith("2") ) return; // this agent is passive
21
22    IRequest r = base.launch(
23       "RANDOM",new Talker( job, "2" ), null );
24    while( r.getStatus() == IRequest.WAITING )
25    {
26       try { Thread.currentThread().sleep(100); }
27       catch( Exception e ) {}
28    }
29    if( r.getStatus() != IRequest.DONE ) return;
30
31    Address a = (Address)r.getInfo("address");
32    r = fireMessage( a, "test", "How are you?" );
33    while( r.getStatus() == IRequest.WAITING )
34    {
35       try { Thread.currentThread().sleep(100); }
36       catch( Exception e ) {}
37    }
38    if( r.getStatus() == IRequest.DONE )
39       System.out.println( "Answer: "+r.getInfo("reply") );
40 }
41
42 /**
43 * Handles message type "test" answering always with the String object
44 * "Fine thanks."
```

```
45  */
46  public boolean handleMessage( Message m, Object object ) {
47
48     if( super.handleMessage( m, object ) ) return true;
49
50     if( m.getType().equals("test") )
51     {
52        System.out.println("Received: "+object);
53        m.setReply("Fine thanks.");
54        return true;
55     }
56
57     return false;
58  }
59
60
61  }
```

G Launch

```
1  import drm.core.Node;
2  import drm.util.ConfigProperties;
3  import drm.agentbase.IRequest;
4  import drm.agentbase.Address;
5  import drm.agentbase.IAgent;
6  import drmhc.HillClimber;
7  import java.util.*;
8  import java.net.InetAddress;
9
10 public class Launch implements Runnable {
11
12
13 // =========== Private Fields ======================================
14 // ================================================================
15
16
17 private Node node = null;
18
19 private final String exper = "hillclimb"+System.currentTimeMillis();
20
21
22 // =========== private methods =====================================
23 // ================================================================
24
25
26 private Address launch( String name, Address root ) throws Throwable {
27
28    IAgent a = new HillClimber( exper, name, root );
29
30    IRequest r = node.launch("RANDOM", a, null );
31    while( r.getStatus() == IRequest.WAITING )
32    {
33       try { Thread.currentThread().sleep(10); }
34       catch( Exception e ) {}
35    }
36
37    if( r.getThrowable() != null ) throw r.getThrowable();
38    return (Address)r.getInfo("address");
39 }
40
41
42 // =========== Public constructors =================================
43 // ================================================================
44
```

```
45
46  public Launch( Node node ) {
47
48     this.node = node;
49  }
50
51  // =========== Public Runnable implementation ========================
52  // ====================================================================
53
54  public void run() {
55  try
56  {
57     System.err.println("Launching hillclimber job");
58     if( node == null ) return;
59
60     // loads the config pars as a system resource from the
61     // classpath which is the jar or directory of the job
62     Properties conf = new Properties();
63     ClassLoader cl = getClass().getClassLoader();
64     try
65     {
66        conf.load( cl.getResourceAsStream(
67           "config.properties" ) );
68     }
69     catch( Exception e )
70     {
71        System.err.println(
72           "Error loading config, using defaults.");
73     }
74
75     // --- launching root
76
77     Address root = launch("root",null);
78     // this should be a local launch
79
80     if( ! root.isLocal() )
81     {
82        System.err.print("Root launched to remote node.");
83        System.err.print(
84           " This is not necessarily what you want.");
85        System.err.println(" Something is going wrong...");
86     }
87
88     // --- launching the rest
89
90     int i=Integer.parseInt(
```

```
91          conf.getProperty("launch.contributors","4"));
92       while( i>0 )
93       {
94          try{ launch( ""+i, root ); }
95          catch( Exception e ) { ++i; }
96          --i;
97       }
98    }
99    catch( Throwable e )
100   {
101      e.printStackTrace();
102   }
103   finally
104   {
105      System.err.println("Launching finished.");
106   }
107
108   }
109
110   }
111
```

H drmhc.HillClimber

```
1  package drmhc;
2
3  import drm.agentbase.Address;
4  import drm.agents.ContributorAgent;
5  import drm.core.*;
6
7  import java.util.List;
8  import java.util.Properties;
9  import java.io.FileWriter;
10
11 public class HillClimber extends ContributorAgent {
12
13 private double currentSolution;
14
15 private double currentValue;
16
17 private int evals = 0;
18
19 private transient Properties conf;
20
21 // ----------------------------------------------------------------
22
23 /** calls super constructor */
24 public HillClimber( String job, String name, Address root ) {
25
26    super( "HillClimber", job, name, root );
27    currentSolution = Math.random();
28    currentValue = Algorithm.eval(currentSolution);
29 }
30
31 // ----------------------------------------------------------------
32
33 /**
34 * Runs the hillclimber. The optimized function is defined in
35 * {@link Algorithm}. The maximal function evals is given by property
36 * "hillclimber.maxEvals". See default config file
37 * <a href="doc-files/config.properties">here</a>.
38 */
39 public void run() {
40
41    // loads the config pars as a system resource from the
42    // classpath which is the jar or directory of the job
43    // It must be loaded here and not construction time because
44    // after construction the agent might be serialized.
```

```
45      conf = new Properties();
46      ClassLoader cl = getClass().getClassLoader();
47      try
48      {
49         conf.load( cl.getResourceAsStream(
50            "config.properties" ) );
51      }
52      catch( Exception e ) {}
53
54      double x, y;
55      final int maxEvals = Integer.parseInt(
56         conf.getProperty("hillclimber.maxEvals","1000"));
57
58      for(; shouldLive && evals < maxEvals; ++evals )
59      {
60         x = Algorithm.mutate(currentSolution);
61         y = Algorithm.eval(x);
62         if( y >= currentValue )
63         {
64            currentSolution = x;
65            currentValue = y;
66         }
67      }
68
69      if( !name.endsWith("root") ) suicide();
70   }
71
72   // ---------------------------------------------------------------
73
74   /**
75   * Returns the current best solution.
76   */
77   public Object getContribution() {
78
79      return new double[] { currentSolution, currentValue };
80   }
81
82   // ---------------------------------------------------------------
83
84   /**
85   * Checks if there is a better solution than our current best.
86   * If there is, we adopts it.
87   * If this is the roor than it attempts to log the contributions to a
88   * file name, which is given as property "hillclimber.outFile".
89   * See default config file
90   * <a href="doc-files/config.properties">here</a>.
```

```
*/
public void collectiveUpdated( ContributionBox peer ) {

   List peers = collective.getContributions();

   for( int i=0; i<peers.size(); ++i )
   {
      double[] contrib = (double[])
         ((ContributionBox)peers.get(i)).contribution;
      if( contrib[1] > currentValue )
      synchronized(this)
      {
         currentSolution = contrib[0];
         currentValue = contrib[1];
      }
   }

   // --- log stuff if root
   // don't forget: this is an example. Much more clever logging
   // should be used than simply dumping out stuff every time.
   if( name.endsWith("root") )
   try
   {
      FileWriter fw = new FileWriter( conf.getProperty(
        "hillclimber.outFile", "hillclimber.out" ), true );
      for( int i=0; i<peers.size(); ++i )
      {
         double[] contrib = (double[])
          ((ContributionBox)peers.get(i)).contribution;
         fw.write(contrib[0]+" "+contrib[1]+"\n");
      }
      fw.close();
   }
   catch( Exception e ) { e.printStackTrace(); }
}



}
```

I drmhc.Algorithm

```
1  package drmhc;
2
3  /**
4  * This class contains algorithmic components needed by the
5  * hillclimber agent. Don't forget that this is a toy example only.
6  * This class is separated mainly to illustrate that a job can
7  * use many classes, not only the agent class.
8  *
9  * <p>It contains an objective function to be maximized
10 * ({@link #eval(double)}) over the domain [0,1]. It contains
11 * operators that operate on this real domain.
12 */
13 public class Algorithm implements java.io.Serializable {
14
15 public static double eval( double x ) {
16
17    // would be far too fast without waiting
18    try { Thread.currentThread().sleep(100); }
19    catch( Exception e ) {}
20
21    return x*Math.sin(x*50);
22 }
23
24 public static double mutate( double x ) {
25
26    x += (2*Math.random()-1)/3.0;
27    if( x < 0 ) x = 0;
28    if( x > 1 ) x = 1;
29
30    return x;
31 }
32
33 }
34
```

J config.properties

```
# This is the default configuration file. Thus if there is no
# configuration file, these are the values that are used.
# You can copy this file to "config.properties" into the root of the
# jar or directory of the experiment and you can modify the values.


# the number of contributors (agents) to launch, default is 4.
launch.contributors = 4

# the maximal number of hillclimbing steps per agent. Default is 1000.
hillclimber.maxEvals = 1000

# the output file to log results to
hillclimber.outFile = hillclimber.out

```