Macondo user's guide

April 7, 1998

Introduction

Large distributed systems pose several constraints on application programming, like bandwidth, latency and high dinamicity of their components. New programming paradigms are needed in order to deal properly with these issues.

Mobile agents are autonomous programs which move themselves between different hosts in a distributed system in order to perform their tasks. Using mobile agents it is possible to move computations where data reside, thus solving latency and bandwidth problems.

Dinamicity issues in open distributed systems require that different applications can interact and synchronize themselves in an open fashion, without restrictions. Coordination technology, which has been initiated by the Linda language at Yale University, is a research field that deals explicitely with these problems.

Macondo, acronym for Mobile Agents and COrdinatioN in Distributed applicatiOns, is a framework for distributed computing which adopts the previous solutions using the Java language. While different existing tools for mobile agents in Java provide *ad hoc* communication mechanism for agents, Macondo mantains mobility and coordination supports completely decoupled. In order to achieve this separation of concerns Macondo provides a powerful coordination language called MJada, Mobile Jada. MJada can then be used to coordinate multiple threads inside a Java program or multiple distributed programs on the Internet, like Java applets and Macondo mobile agents.

Macondo main concepts

In this section the two main building blocks of Macondo, mobility and coordination, will be explained in detail.

Mobility

Mobile agents applications developed with Macondo are based upon the concepts of *agents* and *places*.

Agents are built extending the `Agent` class or the `Traveller` class. The `Agent` class define the basic behaviour of a mobile agent via the `go()` method, which allows the agent to stop its current execution, migrate to the specified host and resume execution in the new location. The `Traveller` class extends the `Agent` class with the usage of *itineraries*, that are sequences of pairs destination/task where each task is executed at the corresponding destination. This way it is possible to decouple application specific code from mobility primitives, leading to a cleaner design for agents.

A place is the execution environment for agents and it is defined by the `Place` class. Each host where Macondo agents may migrate must have a Place running. A Place instance can be used as a stand-alone server or the class can be extended or incapsulated in order to define more sophysticated execution environments.

Coordination

Coordination technology support in Macondo is represented by the MJada coordination language. MJada extends the coordination model of Linda in several directions

- coordination between agents, either mobile or not, takes place using *multiple nested tuple spaces*, while Linda supports a single tuple space. The coordination structure is thus similar to a `UNIX` file system and operations are provided to let agents navigate the structure;

- MJada consuptive operations, `in()` and `read()`, decouple the tuple request from the corresponding result fetching. This way it is possible to make a request, perform other tasks and then test the result for availability. Result can then be fetched immediatly if available, or blocking the calling thread if unavailable. It is also possible to specify a time-out for waiting the result;

- MJada provides two additional operations, `readAll()` and `inAll()`, used to read and withdraw all tuples in a space that match the given template;

- in order to simplify coordinative computing MJada provides a new programming framework based on tuple collections. A collection is a sequence of tuples with the same structure in a given tuple space. Tuples can be added to the collection and then can be read or withdraw in the same order they were added. These consumptive operations are performed using *iterators*, which are objects that cycle over a collection to get tuples. Two predefined iterators, `ReadIterator` and `InIterator` are provided but more complex iterators can be developed;

MJada is implemented as a set of Java classes and it doesn't introduce any syntactic extension to Java. Programs that use MJada are truly Java programs and doesn't require a preprocessor. MJada is built on top of Jada, a coordination kernel for Java.

Installation

Macondo distribution is made of a single archive `macondo.tar.gz`. In order to install successfully Macondo you have to perform the following steps:

1. get and install the Java Developers Kit version 1.1 or higher;

2. get and install the Jada distribution;

3. choose a directory where to install Macondo, say `MACONDO_DIR`, and unpack the content of the archive file in that directory with

   ```
   tar xvzf macondo.tar.gz
   ```

   The previous command will create a directory `macondo`;

4. add the entry `$(MACONDO_DIR)/macondo/classes` to your `CLASSPATH`;

Macondo software is now properly installed and it is possible to test the installation with programs contained in the `demo` directory.

Managing a Macondo system

A Macondo system needs run time supports for mobile agents, which must be able to migrate between different hosts, and for programs which use MJada to coordinate their activities. In this section these different kinds of support will be described.

Mobility run-time support

The execution environment for Macondo mobile agents is provided by the `Place` class. It can be used as a stand-alone server or it can be extended or incapsulated in other classes. When used as a server it accepts the following command line syntax

```
java Place [-p port] [-v] [-l] [-h[elp]]
```

where the options have the following meaning

- `-p port`
  runs the place at with the specified port number. If this option is not specified, the place will be running at the default port 8000;

- `-v`
  runs the place in verbose mode. In the current version it simply prints a startup message;

- `-l`
  load predefined agents in this place, looking for agents' names in the file `macondo.agents` that must be available in the current directory where the place is started. The file is simply composed of agents' class file name, each on a different line, and code for agents must be available locally;

- `-h, -help`
  prints usage information and exit;

The configuration of a `Place` object is done using Java properties, specified in a file called `macondo.properties`. This file must be available in user's home directory or in the directory where `Place` is launched. The following properties can be specified

- **MACONDO.HOME**: installation directory for the Macondo distribution;

- **DIR.CLASSES**: directory where classes loaded from the net are stored;

- **DBPATH**: directory where object are stored, used by the persistence support of Macondo;

- **STORE.CLASSES**: if this is set to `"true"`, then classes loaded from the net will be stored in the directory specified by the **DIR.CLASSES** property. Storing classes is required in order to allow code migration between multiple places;

MJada run-time support

Clients using the MJada coordination framework have to connect to a tuple space server. Different servers may be running on different hosts, but if the client program is an applet it can only open a connection with the host from which the applet code has been downloaded.

MJada's servers are implemented by the `SpaceServer` class. It is a minimal tuple server which accepts and serves requests from the net. It can be launched with the following syntax

```
java SpaceServer [-p port] [-v] [-h[elp]]
```

Command line options have the following meaning

- `-p port`
  runs the server with the specified port number. If this option is not specified, the server will be running at the default port 7654;
- `-v`
  runs the server in verbose mode: each request to the server will be displayed;
- `-h, -help`
  prints usage information and exit;

The `SpaceServer` class can also be extended or used in other classes. Clients connect to a `SpaceServer` using a `SpaceClient` object.

Examples

In this section a complete Macondo application will be shown. Two agents, `PingAgent` and `PongAgent`, are created in a place and then migrate to another host called `pingpong`. The place running at the destination host has a tuple space that agents use to synchronize themselves exchanging tuples with an increasing counter for five times.

The application described is very interesting because it shows how to use mobile agents in Macondo and their coordination with MJada.

Now source code for agents is presented. The `PingAgent` and `PongAgent` are simmetrical, except for startup that is managed by the `PongAgent`. The `init()` method, overridden from the `Agent` class, let agents migrate to the host called `pingpong` where a place is running at port 9000. The `play()` method is then executed at the destination host, where the coordination activity takes place.

The code for `PingAgent` and `PongAgent` follows.

```
import macondo.mak.*;
import macondo.mjada.*;

public class PingAgent extends Agent {

    public PingAgent() {}
```

```
    public void init() {
        try {
            go("pingpong", 9000, "play");
        } catch (TransportException e) {
            e.printStackTrace();
        }
    }

    public void play() {
        PingPongPlace place = (PingPongPlace)getCurrentPlace();
        Space space =  place.getSpace();

        Tuple ping, formal = new Tuple("ping", Integer.class);
        Result res;
        int cnt;

        for(int i=0; i<5; i++) {
            res = space.in(formal);
            ping = (Tuple)res.getObject();
            System.out.println("Ping: got "+ping);
            cnt = ((Integer)ping.getItem(1)).intValue();
            space.out(new Tuple("pong", new Integer(cnt+1)));
        }

    }
}

import macondo.mak.*;
import macondo.mjada.*;

public class PongAgent extends Agent {

    public PongAgent() {}

    public void init() {
        try {
            go("pingpong", 9000, "play");
        } catch (TransportException e) {
            e.printStackTrace();
        }
```

```
    }

    public void play() {
        PingPongPlace place = (PingPongPlace)getCurrentPlace();
        Space space =  place.getSpace();

        Tuple pong, formal = new Tuple("pong", Integer.class);
        Result res;
        int cnt;

        space.out(new Tuple("ping", new Integer(0)));
        for(int i=0; i<5; i++) {
            res = space.in(formal);
            pong = (Tuple)res.getObject();
            System.out.println("Pong: got "+pong);
            cnt = ((Integer)pong.getItem(1)).intValue();
            space.out(new Tuple("ping", new Integer(cnt+1)));
        }

    }
}
```

The `BootPlace` extends Macondo `Place` class in order to provide a customized environment. Its only task is to launch the agents, providing also to migration issues.

```
import macondo.mak.*;

public class BootPlace extends Place {

    public static void main(String[] args) {
        Place place = new BootPlace();
        place.setPort(8000);
        place.startPlace();

        try {
            place.launchAgent( new PingAgent() );
            place.launchAgent( new PongAgent() );
        } catch (Exception e) {
            System.out.println("Agents can't be launched");
            e.printStackTrace();
```

```
        }
    }

}
```

The previous code must be available on the host where agents will be launched. It can be compiled with

```
$ javac PingAgent.java
$ javac PongAgent.java
$ javac BootPlace.java
```

The `PingPongPlace` class is a more advanced place which provides a tuple space for agents interactions.

```
import macondo.mak.*;
import macondo.mjada.*;

public class PingPongPlace extends Place {

    private Space space;

    public PingPongPlace() {
        super();
        space = new Space();
    }

    public Space getSpace() {
        return space;
    }

    public static void main(String[] args) {
        PingPongPlace place = new PingPongPlace();
        place.setPort( 9000 );
        place.startPlace();
    }

}
```

The code must be available on the host called `pingpong` and can be compiled with

```
$ javac PingPongPlace.java
```

The place can then be started with

```
$ java PingPongPlace
```

Agents are launched on a different host with

```
$ java BootPlace
```

When agents will reach the host `pingpong` they will display messages on their activity.

It's worth noting that agents' source code need not to be available on the host which is running the `PingPongPlace` object: it will request the code to the `BootPlace` object.