ActorFoundry: A Tutorial Guide

Sarmad Abbasi and **Zubair Zahid**[1]
Department of Computer Science
Sukkur Institute of Business Administration
Sukkur 65200, Sindh
Pakistan

[1]This work is a part of the authors final year project.

Contents

Chapter 1

ActorFoundry Basics

1.1 Introduction

Actors are independent agents for distributed computing[1]. An actor is a concurrent and autonomous entity that has a unique immutable address. Furthermore, each actor has its own mutable local state. Actors communicate by sending messages asynchronously. They are a flexible and powerful model for distributed computing and have been implemented on several platforms[1].

ActorFoundry is a JVM-based framework for Actor programming[2]. It enables writing actor programs in familiar Java syntax. There is a vast literature available on Actors and we recommend that before continuing this tutorial you read the recent paper by Agha and Karmani [1] that covers the basics of actors, discusses various applications and provides a comprehensive list of actor bases systems that are available.

This tutorial is a step by step guide to `ActorFoundry`. It is written with a specific purpose in mind: to enable students studying at Sukkur IBA to with a clear and accessible introduction to ActorFoundry that would allow them to undertake final year projects that use `ActorFoundry`. We hope that the tutorial is also helpful to for others trying to learn the basics of `ActorFoundry` and actors. The pre-requisites for this tutorial are:

1. A through reading of the two papers [1] and [2].
2. Familiarity with Java programming. It is still possible to follow most of the tutorial if you are familiar with C^{++}; indeed the first author developed this tutorial while learning Java.
3. A course in distributed computing or distributed algorithms.

If you wish to learn programming using the ActorFoundry and work on actor based system, we suggest you give the paper by Agha and Kamrani [1] a first reading. You may also want to look at [2]. Then you can start this tutorial. The ideal way to learn is to go back and forth between the tutorial and these papers: thus the purpose of this tutorial is to combine practice with theory. This, hopefully, would enable people to learn Actors conceptually and be able to use them to write concrete applications in ActorFoundry.

1.2 Installing ActorFoundry, Compiling and Running Examples

If you wish to work with ActorFoundry, we recommend the Linux Operating System. ActorFoundry easily installs on Linux and one can cut through a lot of nonsense and start working in it. The second option is to run it on a Mac (almost as good). The last option (guess) is Windows. We discuss how to install the system on these three platforms.

1.2.1 Linux

Setting up ActorFoundry on the Linux (we used Ubuntu 10.04 LTS) is just a work of few steps. Linux saves the trouble of setting up environment variables and defining directory paths as on Windows. So Linux is recommended if you really want to do a serious programming. Following are the steps to setup the ActorFoundry:

1. Download "`foundry-local-src-1.0.tar.gz`" from `http://osl.cs.uiuc.edu/af/`
   Uncompress and unzip the file which will create your main foundry directory.

2. Install `openjdk-6-jdk` package from Ubuntu software center OR type "`sudo apt-get install openjdk-6-jdk`" in terminal.

3. Install `ant` from Ubuntu software center OR type "`sudo apt-get install ant`" in terminal.

   ```
   cd ~/Desktop/foundry-local-src-1.0
   ```

   will take you to the main foundry directory.

4. Type "`ant`" to compile the initial files.

Thats it, we are ready :)

1.2.2 Mac

Setting up ActorFoundry on a Mac is also not difficult. However, we were only able to do it successfully on machines that were running "snow leopard." Along with snow leopard you can install Xcode that comes standard with the snow leopard operating system. That contains JDK 6. You may also have to do some system updates to make sure you have the correct version of JDK. Following are the steps to setup the ActorFoundry:

1. Download "`foundry-local-src-1.0.tar.gz`" from `http://osl.cs.uiuc.edu/af/`
   Uncompress and unzip the file which will create your main foundry directory. Open a terminal and go to the main foundry directory. If you unzipped the foundry on your desktop then:

   ```
   cd ~/Desktop/foundry-local-src-1.0
   ```

   will take you to the main foundry directory.

2. Type "`ant`" to compile the initial files.

Thats it, we are ready :)

1.2.3 Windows

To be written.

1.2.4 Running `Hello World`

The command:

```
java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart osl.examples.helloworld.HelloActor hello
```

Will run the `Hello World` program. If everything worked perfectly, you should have something like this printed on the screen!

```
BasicActorManager v1.5, ActorFoundry v1.0
Hello World!
Exiting...
```

If this is printed on the screen, you should jump in joy. Congratulations! The foundry has been installed properly and you have just run your first *actor* program. Now, you are ready to begin actor programming using the ActorFoundry.

Mac Users: In case the program does not print `Hello World` on the screen then try the command:

```
java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart osl.examples.helloworld.HelloActor hello 1
```

which just appends a parameter to the previous command. This should fix the problem. We think this problem is due to some bug in parameter passing in ActorFoundry and needs further investigation. You only have to modify this command once. The rest of the tutorial remains the same for both Linux and Mac users.

Let us look at this command and break it up and see what exactly happened. `java` is the java application launcher. The `-cp lib/foundry-1.0.jar:classes` option specifies the class path. The next argument `osl.foundry.FoundryStart` runs the ActorManager. The actor program that we wish to run is `osl.examples.helloworld.HelloActor` and `hello` is the message that we want to send to this actor, when it starts. For the rest of the tutorial the following command would run an actor program:

`java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart` *ActorClass messag [parm]*

You can interpret this command as saying that create an actor from the class `ActorName.class` and send it a message *messag* with, possibly optional, parameters *pram*. This gets an actor program going.

Remark: We have found a bug in how parameters are passed in ActorFoundry. Therefore, if you plan to write programs that require more than one parameter to be passed, be very careful.

1.3 Understanding Hello World

The most important task is now to understand the various components of the `Hello World` program. The source code is in the directory `src/osl/examples/helloworld`. When you go to that directory you would find three files. Let us open the file `HelloActor.java` and examine it. This tutorial will only show the relevant parts of the code in these programs. Comments and other parts of the code that are not necessary to develop the understanding needed are ignored. The beginning of the file contains the following statements:

```
package osl.examples.helloworld;
import osl.manager.Actor;
import osl.manager.ActorName;
import osl.manager.RemoteCodeException;
import osl.manager.annotations.message;
```

The first statement specifies that this code is a part of the package `osl.examples.helloworld`. The import statements import various things needed to write an actor program. These are:

1. `osl.manager.Actor`: All actors extend the class `Actor`. Thus this import is essential for any actor program.

2. `import osl.manager.ActorName`: This import is needed if the actor program creates a new actor or refers to another actor. The class `ActorName` is capable of storing the *immutable address* of another actor . This import is not necessary for actors that do not refer to other actors.

3. `import osl.manager.RemoteCodeException`: This import is needed when an actor creates a new actor. The manager may fail to create a new actor and an exception is thrown. This import is also needed when a `call` is made to another actor. An exception is thrown if a `call` to another actor fails.

4. `import osl.manager.annotations.message`: This import is needed for all actor programs receive or send a message. Thus this import is needed for all actor that do anything at all.

For the rest of the tutorial these statements will not be considered when we refer to fragments of code. However, we urge you to look at the import statement and see why they are needed in a particular actor program. Next there is the class `HelloActor` which is defined in the file `HelloActor.java`

```
public class HelloActor extends Actor {
  @message
  public void hello() throws RemoteCodeException {
    ActorName other = null;
    call(stdout, "print", "Hello ");
    other = create(WorldActor.class);
    send(other, "world");
  }
}
```

The `@message public void hello()` specifies the behavior of the `HelloActor`, when it receives the message “`hello`.” This actor prints `Hello` on the screen (This is actually done by calling the actor `stdout`, but we will not bother with that now), then it creates an actor from `WorldActor.class`. The last step is to send the message `world` to this newly created actor. Notice that the *immutable* name of newly created is stored in the variable `other`. Which is then used to send a message to this `WorldActor`. Let us open the file `WorldActor.java` to see the actor class `WorldActor`.

```
public class WorldActor extends Actor {
 @message
  public void world() {
    send(stdout, "println", "World!");
  }
}
```

This actor upon receiving the message `world` simply prints `World` on the screen! Thus our command

```
java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart osl.examples.helloworld.HelloActor hello
```

Creates a `HelloActor` and sends it the message `hello`. The `HelloActor` upon receiving the message `hello` prints `Hello` on the screen and then creates an actor from the class `WordActor` and send it the message `world`. This `WorldActor` upon receiving the message `world` prints `World` on the screen.

Lastly, the `ActorManager` then realizes that there are two actors both now waiting for messages. There are no unprocessed messages in the system. Thus, it concludes that there can be no further activity in the system and exits the program.

Exercise 1.1 *Type the command:*
`java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart osl.examples.helloworld.HelloBoot boot`
and see the output. Then examine the file `HelloBoot.java`*. Can you explain why the output is exactly the same as with the previous command?*

Exercise 1.2 *The command:*
`java -cp lib/foundry-1.0.jar:classes osl.foundry.FoundryStart osl.examples.helloworld.HelloActor hello`
is quite cumbersome to type. Make a file called `runnow` *and store the command there. Type*
`chmod +x runnow`
in the terminal to make the file executable. You can just type ./runnow to run the `HelloWorld` *program. This file can be modified to run other programs also.*

1.3.1 A note on terminology

Consider the statement

```
other = create(WorldActor.class);
```

We can also consider the following statements that can potentially appear in an actor program.

```
other1 = create(WorldActor.class);
other2 = create(WorldActor.class);
```

The file "`WorldActor.java`" contains the definition of the actor class `WorldActor`. Two actors are created and their names are stored in the variables `other1` and `other2`. When the file `WorldActor.java` is compiled it creates a file called `WorldActor.class`. We will sometimes refer to these two actors by the variables that store their immutable name. However, sometimes when there is only one actor that is created from a particular class we may just call it by the name of the class.

1.4 Modifying `Hello World` I

In this section, we will make simple modifications to the `Hello World` program. Let us consider the following problem:

Problem 1 *Modify the* `Hello World` *program so that it prints* `Hello World` *and* `Goodbye World` *by adding a new actor class called* `GoodbyeActor.java` *to the code. You may make some minimal changes to* `HelloActor.java` *and* `WorldActor.java`*.*

You will find the solution to this problem in the directory `src/osl/tutorial/hello_goodbyeI`. You should go to this directory and examine the files present there. The story of this play is that there are four actors: `BootActor, HelloActor, WorldActor` and `GoodbyeActor`.

The `BootActor` is created from the `Boot_I.class` by the Foundry Manager. `BootActor` on the receiving the message `boot` will create `HelloActor` from `HelloActor_I.class` and will send a `hello` message to `HelloActor`. Following code is from the file `Boot_I.java` that defines the actor class `Boot_I`.

```
public class Boot_I extends Actor {
 @message
    public void boot() throws RemoteCodeException {
    ActorName HelloActor = create(HelloActor_I.class);
    send(HelloActor, "hello");
  }
}
```

`HelloActor` upon receiving the message `hello` will print "`hello`" on the screen, create `WorldActor` from `WorldActor_I.class` and will send a `world1` message to `WorldActor`. The file `HelloActor_I.java` defines the actor class `HelloActor_I`.

```
public class HelloActor_I extends Actor {
  @message
   public void hello() throws RemoteCodeException {
    ActorName WorldActor = create(WorldActor_I.class);
    call(stdout, "print", "Hello ");
    send(WorldActor, "world1");
  }
}
```

`WorldActor` upon receiving the message `world1` will print "`world`" on the screen, create `GoodbyeActor` from `GoodbyeActor_I.class` and send a `goodbye` message to `GoodbyeActor`. While on the reception of message `world2`, it will only print "`world`" on the screen. Note that upon receiving the message `world2`, the `WorldActor` *does not* create any actor.

Following is the relevant code from `WorldActor_I.java` that defines the actor class `WorldActor_I`.

```
public class WorldActor_I extends Actor {
        @message
  public void world1() throws RemoteCodeException {
    ActorName GoodbyeActor = create(GoodbyeActor_I.class);
    send(stdout, "println", "World!");
    send(GoodbyeActor,"goodbye");
  }
        @message
  public void world2() {
    send(stdout, "println", "World!");
  }
}
```

GoodbyeActor upon receiving the message goodbye will print "goodbye" on the screen, create WorldActress from WorldActor_I.class and send a world2 message to WorldActress. Following is the relevant code from GoodbyeActor_I.java.

```
public class GoodbyeActor_I extends Actor {
        @message
  public void goodbye()throws RemoteCodeException {
    ActorName WorldActress = create(WorldActor_I.class);
    send(stdout, "print", "Goodbye ");
    send(WorldActress,"world2");
  }
}
```

Note: In this program *two instances* from the WorldActor_I.class are created. One that we call WorldActor, which is created by the HelloActor. The other is called WorldActress, which is created by the GoodbyeActor.

Exercise 1.3 *In the above solution two instances of the* WorldActor_I.class *were created. We want to now, come up with a solution that only creates one such instance to which the* GoodbyeActor *sends a message. This problem is solved for you in* src/tutorial/hello_goodbyeI_ex*. Examine, all the files placed in the directory and understand the solution completely. Note how an Actor can send its own name to another Actor using* self()*.*

1.5 Modifying `Hello World` II

In this section, we will further understand simple programming with actors. We will see how two Actor programs can behave the same way, while having entirely different internal structures. Our goal is to write two actor programs that perform a simple task.

Problem 2 *Write an actor program that print "*`Hello World`*" exactly n times on the screen, where n is passed to the* `BootActor` *as a command line argument. It should create n instances of the* `HelloActor` *and n instances of the* `WorldActor`*. Each* `HelloActor` *should print "*`Hello`*" exactly once on the screen and each* `WorldActor` *should print "*`World`*" exactly once. n would be passed to the* `BootActor` *as an command line argument.*

The solution is provided in the directory `src/tutorial/hello_worldII`. You should look at the files in that directory. We have modified the original hello world program. Firstly, the `BootActor`, must be able to accept a parameter and it has to pass it on to the `HelloActor`.

The `BootActor` will receive a parameter n via message boot. It will pass this parameter to the `HelloActor` that it creates.
The code is from the file `bootActor_II.java` that defines the actor class `bootActor_II` is given below:

```
public class bootActor_II extends Actor {
@message
  public void boot(Integer n) throws RemoteCodeException {
    ActorName helloActor = create(HelloActor_II.class);
    send(helloActor, "hello",n);
  }
}
```

The basic idea is that the `helloActor` will now receive a message along with a parameter n. It will assume the additional responsibility of passing this parameter to the `WorldActor` that it creates.
Following code is file `HelloActor_II.java` that defines the actor class `HelloActor_II`.

```
public class HelloActor_II extends Actor {
  @message
    public void hello(Integer n) throws RemoteCodeException {
    ActorName worldActor = null;
    call(stdout, "print", "Hello ");
    worldActor = create(WorldActor_II.class);
    send(worldActor, "world",n);
  }
}
```

Now the `worldActor`! It receives a parameter n. By examining the parameter it decides whether to create another HelloActor or not. If $n > 1$ then it needs to create a new `HelloActor` and pass $n-1$ to it as a parameter. Following code is from the file `WorldActor_II.java` that defines the actor class `WorldActor_II`.

```
public class WorldActor_II extends Actor {
@message
  public void world(Integer n) throws RemoteCodeException {
      send(stdout, "println", "World!");
  if (n > 1) {
  ActorName helloActor = create(HelloActor_II.class);
  send(helloActor,"hello", n-1);
  }
  }
}
```

Next, we would accomplish the same task by creating only one copy of the `HelloActor` and one copy of `WorldActor`. Note that the above solution almost works. Now, the `WorldActor` does not create a new `HelloActor` but only can send a message to the HelloActor that created it. However, for that it needs the the address of the `HelloActor`. This can be accomplished by using `self()`.

Exercise 1.4 *Write another program that will accomplish the same task by creating only one copy of the* `HelloActor` *and one copy of the* `WorldActor`*.*

1.6 To `call` or to `send`

In this section, we illustrate an important difference between two ways of sending messages: The first one is `send` and the second one is `call`. You must have noticed that to print, say `Hello` on the screen we use:

```
call(stdout,"print", "Hello");
```

This has the same effect as the usual java way of printing on the screen; that is, when we use

```
System.out.print("Hello");
```

What we are actually doing in ActorFoundry is sending a message to an actor. The actor is `stdout`. The message is `print` and the parameter is `"Hello"`. What does `call` exactly mean? Since, actors may process messages in *any order* does that mean that

```
call(stdout,"print", "Hello ");
call(stdout,"print", "World ");
```

can produce

```
World Hello
```

as an output? The answer is no! The semantics of `call` specify that the caller cannot proceed with the remaining code till the callee has finished processing the message that was sent to it. Thus, the message `call(stdout,"print", "World")` is not sent until `stdout` has processed the first message; that is, processed `call(stdout,"print", "World")` and printed "`hello`" on the screen. This is very useful in many contexts and makes actor programming simpler. `call` can be used to *synchronize* actors. Though, theoretically it is possible to replace all `call` statements by *asynchronous* messages, as specified in the original actor semantics. The `call` facility makes the code conceptually easier to understand. The usual way of sending messages *asynchronously* is done using `send`. Thus,

```
send(stdout,"print", "Hello ");
send(stdout,"print", "World ");
```

can in principle produce the output

```
World Hello
```

at times (but probably in practice that happens very rarely). The difference between `call` and `send` is illustrated in the directory `src/tutorial/CallSendDemo`. Let us look at actor class `CSHelloActor` code in file `CSHelloActor.java`.

```
public class CSHelloActor extends Actor {
  @message
  public void hello() throws RemoteCodeException {
    ActorName WActor = create(CSWorldActor.class);
    call(stdout, "print", "Hello ");
    call(stdout, "print", "(calling world actor) ");
    call(WActor, "world");
    call(stdout, "print", "Goodbye....\n");

    call(stdout, "print", "Hello Again! ");
    call (stdout, "print", "(sending world actor) ");
    send(WActor, "world");
    call(stdout, "print", "Goodbye and this time I mean it ");
  }
}
```

This actor creates a `WActor` from the actor class `CSWorldActor.class`. It then prints "`Hello,`" prints "`(calling world actor)`" and then *calls* the `WActor` to print "`World`" and then prints "`Goodbye....`" Note that since it calls the `WActor`, therefore, no matter how busy the `WActor` is, this actor will wait till the `WActor` responds (which will mean that the `CSWorldActor` has printed "`World`" on the screen) before printing "`Goodbye....`"

On the other hand, the last four lines do a similar thing. However, this time the message is delivered to the `WActor` using `send`. Thus, `CSHelloActor` would not wait for `WActor`. If the `WActor` is busy then "`Goodbye and this time I mean it` " will be printed before `World`. Lets look at the `CSWorldActor.java` and find out how busy it is!

```
public class CSWorldActor extends Actor {
        @message
  public void world() {
    try {
    Thread.sleep(4000);
     } catch(Exception ie){};
    send(stdout, "println", "World!");
  }
}
```

Well, this actor takes his time before printing `World`. Thus, when it is called second time the result it that `Goodbye and this time I mean it` is printed before `World`. You should run this code, examine all the files and understand clearly the difference between `send` and `call` before proceeding.

Next we consider the following problem:

Problem 3 *Write a program to print the Bertrand Russell's quote, using five actors,*

1. `bootActor` *which sends boot message to activate other four actors.*
2. `russelActor` *which prints* `"Three passions, simple but overwhelmingly strong, have governed my life:"`
3. `loveActor` *which prints* `"the longing for love,"`
4. `knowledgeActor` *which prints* `"the search for knowledge,"`
5. `pityActor` *which prints* `"and unbearable pity for the suffering of mankind."`

So the combined output should be:

```
Three passions, simple but overwhelmingly strong, have governed my life: the longing for love,
the search for knowledge, and unbearable pity for the suffering of the mankind.
```

The last four actors upon receiving appropriate messages and an integer parameter `t` *should sleep for* `t` *milliseconds before printing on the screen.*

The solution is given in `src/tutorial/RussellQuote`. The `thoughtBoot` actor on the reception of `boot` creates and activates other four actors using different messages for each actor.
Following is the relevant code from `ThoughtBoot.java`.

```
public class ThoughtBoot extends Actor {
  @message
  public void boot() throws RemoteCodeException {
  ActorName russelActor = create(RusselActor.class);
  ActorName loveActor = create(LoveActor.class);
  ActorName knowActor = create(KnowActor.class);
  ActorName pityActor = create(PityActor.class);
```

```
  call(russelActor, "think",0);
  call(loveActor, "love",7000);
  call(knowActor,"knowledge",5000);
  call(pityActor, "pity",3000);
}
```

The `russelActor` which on the reception of `think(Integer t)` message prints "`Three passions, simple but overwhelmingly strong, have governed my life:`" after sleeping for `t` milliseconds. Following is the relevant code from the file `RusselActor.java` that defines the actor class `RusselActor`:

```
public class RusselActor extends Actor {
  @message
  public void think(Integer t) throws RemoteCodeException {
    try{ Thread.sleep(t); }
    catch(InterruptedException ie){ }
    call(stdout, "println", "Three passions, simple but overwhelmingly strong, have governed my life
  }
}
```

The other three actors are very similar. You can examine the relevant files and understand the program without difficulty.

Exercise 1.5 *Make the following changes in the* `ThoughtBoot.java`*.*

1. *Change time parameter for the four calls. Observe that the quote printed is still exactly the same. Understand why that is the case.*
2. *Now, replace the four calls with four* `send` *operations. Try to change the dialogue timing of last three actors in such a way that you get all six possible permutations of (grammatically incorrect) quotes. You should get al the permutations by changing the times and not the order of the statements in the code.*

Hint: Dialogue timings are controlled by `sleep()` method.

Chapter 2

Intermediate Skills

2.1 Introduction

This chapter will allow you to reach intermediate level.

2.2 Leader Election in a Ring: A complete example

Leader election is an important problem in distributed Algorithms. In this section, we demonstrate a simple leader election algorithm and implement it using `ActorFoundry`. The algorithm we have chosen is conceptually very easy to understand. The purpose of this demonstration is to clarify certain concepts in distributed algorithms and see a complete example that is worked out using `ActorFoundry`.

The algorithm assumes that there are n processors $p_0, \ldots, p_{n-1}$ arranged in a ring. Each processor has a unique id and can send a message to the next processor. Thus p_i can send a message to $p_{i+1 \bmod n}$. However, the processors are unaware of n, the total number of processors in the ring. The task is to elect a leader amongst these n processors via a distributed algorithm without centralized control.

The algorithm is described in Nancy Lynch's book[3]. The solution is simple and intuitive. All the processors send their id to the next processor. From then on each processor keeps track of the minimum id seen so far. All ids larger than the minimum are ignored. Whenever a processor receives an id that is smaller than the minimum, the processor updates its minimum and passes that id to the next processor in the ring. The processor that receives its own id back becomes the leader. The leader can then initiate any other algorithm. In this example, the leader then initiates an algorithm that assigns each processor its position in the ring.

Exercise 2.1 *Write a program called* `RingDemo` *that will implement the leader election algorithm on a ring using* `ActorFoundry`*. Your program should contain a* `RingBoot` *Actor that is used to start the system. This boot actor takes a parameter n. The boot actor then creates n identical actors. It further supplies each actor with:*

1. *A random ID between* 0 *and* $n-1$*.*
   Note that the actor name can serve as a unique id. However, here we are insisting that the boot actor supplies each actor with a unique ID.

2. *The immutable address of its next neighbor.*

After that the boot actor sends a message `electleader` *to all the actors. These actors should then simulate the "leader election" algorithm given in [3]. At the end of the algorithm the leader should output its id. The actors should then proceed to rename themselves from* 0 *to* $n-1$ *in a circular order starting from the leader in the ring. Each actor should output its old id and its new id (that is, the position in the ring).*

2.3 The implementation

The implementation consists of two programs. The first one is the boot actor that "sets up the scenario." The boot actor creates n actors and sends each actor a unique id from $\{0, \ldots, n-1\}$. The ids are shuffled randomly. It also sends each actor the name of its next neighbor in the ring. Finally it signals each actor to initiate the leader election algorithm. The directory `src/osl/tutorial/RingDemo` contains the solution. Let us look at the file `RingBoot.java` that defines the actor class `RingBoot`:

```
public class RingBoot extends Actor {
Random generator = new Random();

    @message
        public void boot(Integer n) throws RemoteCodeException  {
        Integer PId[]= new Integer[n];
        for(Integer i=0; i < n; i++) PId[i] = i;
        for(Integer i = n-1; i> 0; i--)
            {   Integer j = generator.nextInt( i+1 );
                Integer temp = PId[j];
                PId[j] = PId[i];
                PId[i] = temp;
            }
        ActorName RingActor[] = new ActorName[n];
        for(Integer i=0; i < n; i++)
                RingActor[i] = create(NodeActor.class);
        for(Integer i=0; i < n; i++)
                call(RingActor[i],"next",RingActor[(i+1)%n]);
        for(Integer i=0; i < n; i++)
                send(RingActor[i],"activate",PId[i]);
    }
}
```

The first two loops are a standard way to create a random permutation of ids from the set $\{0, \ldots, n-1\}$. Then n actors are created and each one is sent the name of its next neighbor in the ring. Note, that we use `call` to communicate the address of the next actor. This is done, because we want to make sure that before initiating the leader election algorithm, each actor knows the immutable address of the next actor. The last loop initiates the leader election algorithm.

Let us look at the file `NodeActor.java` to see how the n newly created nodes would behave. The members `myid`, `min` and `RingPos` contain the id of the actor, the minimum id seen while running the leader election algorithm and `RingPos` is the position of this actor in the ring. This is a quantity that needs to be computed. The `Boolean` variable `leader` should be true at the end of the algorithm for exactly one actor. `NextActor` is used to send messages to the next actor in the ring.

```
public class NodeActor extends Actor
{   Integer myid ;        // Actor Uid
    Integer min;        // The minimum Id recieved.
    Integer RingPos;    // The Ring Position of this ID. To be computed.
    Boolean leader=false;
    ActorName NextActor; // The Next Actor in the Ring
```

Now, let us look at the various messages:

```
  @message // This message recieves actor address and assigns it to NextActor
        public void next(ActorName A) { NextActor = A; }
```

Upon receiving the name of the next actor; the actor records it in the variable `NextActor` for future use.

```
@message
      public void activate(Integer i)
  {   min= myid = i;
      send(NextActor,"newId",myid);
  }
```

Upon receiving the `activate` message. Each actor sets its own id and `min` to be its own id, since it has not yet seen any other ids. It passes its own id along in the ring.

```
@message
      public void newId(Integer NId) {
          if (NId < min)
            { min = NId;
              send(NextActor,"newId", min);
            }
          else if (NId == myid)
            { leader = true;
              RingPos = 0; // The Leader sets the Ring Postion to 0
                           // and initiates others to set their Postion
              send(NextActor,"SetRingpos",RingPos+1);
            }
  }
```

Every time a processor receives a `newId` message along with a id. It compares that id with the current minimum. If the new id is small than the minimum, the minimum is updated and passed forward. In case, it is not smaller than the minimum, the the newly received id is compared with the current processors id. If it matches then the processor realizes that it is the leader. It sets it own `RingPos` to 0 and initiates a `SetRingpos` algorithm.

Exercise 2.2 *The* `SetRingPos` *and* `PrintRingPos` *messages are not described here, yet they are a part of* `NodeActor.java`*. Understand clearly what they do.*

Acknowledgments

We would like to thank Gul Agha and Rajesh Karmani for extending all kinds of help while we were working on this tutorial. They have been very generous with their time and have answered all our queries almost instantaneously. We would also like to thank our colleagues Nahdia Majeed and Fabeha Shamsi for meticulously going through this tutorial. They have given us constant feedback that has allowed to remove numerous mistakes and oversights.

Bibliography

[1] Rajesh Kumar Karmani and Gul Agha. Actors. In *To appear in Springerś Encyclopedia of Parallel Computing*, 2010.

[2] Rajesh Kumar Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A comparative analysis. In *The proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2009.

[3] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.