Parallel and Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

LAB DISTRIBUTED ALGORITHMS (IN4150)

Java/RMI Manual

D.H.J. Epema and M.N. Yigitbasi

January 28, 2011

In this document we explain how distributed algorithms and systems can be implemented in an object-oriented way using Java and Java Remote Method Invocation (RMI). RMI is for an object-oriented language like Java what RPC is for an imperative language like C. As a result, many of the things a programmer has to do to get a running Java/RMI program are the same as in the case of a C/RPC program, such as creating an interface and producing the client and server stubs with an interface compiler.

It is assumed that the reader has a basic knowledge of how to write single-machine Java programs. In Section 7, you will find a list of references to web sites giving more detail about Java and Java RMI.

1 Java RMI

Normally, objects can only call methods of objects residing in the same Java Virtual Machine (JVM). In order for the methods of objects in a Java program to be called remotely, that is, across the boundaries of JVMs, the programmer has to explicitly express this. For this purpose, Java defines an interface (in the technical Java sense) `Remote`, that does not contain any methods. It is up to the programmer to declare an interface that extends `Remote`, and a class that implements that interface. The methods of this class can then be called remotely. In addition, objects making a remote invocation have to handle specific remote exceptions. This means that RMI is non-transparent to both the object that is called remotely, and to the object doing the remote call.

Remote Interfaces

A remote interface has to extend the standard interface `Remote`, and has to declare the remotely accessible methods in the following way:

```
public interface My_interface_RMI extends Remote {
   public void method1(parameters) throws java.rmi.RemoteException;
   ...
}
```

(Exceptions related to RMI are explained below.) Which methods are defined in the interface depends completely on the application.

Remote Classes

A class with remote methods has to extend the class `RemoteUnicastObject`, which is a predefined Java class for objects that live only for the duration of the program implementing the remote object, and has to implement a remote interface as defined above. Its declaration should be of the form

```
public class My_class extends UnicastRemoteObject
                      implements My_interface_RMI {
   variable and method declarations
}
```

Importing Packages

The packages `java.rmi.*` and `java.rmi.server.UnicastRemoteObject` contain classes related to RMI. The first of these has to be imported into every file in which a remote interface or class is defined or used. The second has to be imported into the file defining a remote class.

Exceptions

Exceptions in programming languages are used to catch and report errors. These errors may be of very different natures, such as dividing by zero, trying to access a non-existent file, or trying to invoke a method of a non-existent remote object. In particular in distributed programs, many things can go wrong, and hence it is sensible to use the exceptions that may be thrown by remote objects.

In Java there are two types of exceptions: checked and unchecked. Checked exceptions extend the `java.lang.Exception` class and they must be caught (with a try-catch block) or passed further up the call stack (with the `throws` keyword). Examples of checked exceptions are `java.rmi.RemoteException` and `java.net.MalformedURLException`. Unchecked exceptions extend the `java.lang.RuntimeException` class and do not need to be caught, since they are due to errors in the program logic and cannot be reasonably recovered at runtime. Examples of unchecked exceptions are `DivideByZeroException` and `ArrayIndexOutOfBoundsException`.

For the exceptions that may be thrown at RMI runtime, please see

`http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmi-exceptions.html.`

The RMI Registry

In order for an object to call a method of a remote object, the remote object has to be registered with the `rmiregistry` on its local host, and the former has to consult that `rmiregistry` to obtain a reference to the remote object. The name with which an object registers itself has a URL syntax of the form

```
rmi://host:port/object-name
```

where `host` is the internet name of the host[1] on which the object lives, `port` is a port number which may be omitted, and `object-name` is any name chosen for the remote object. For instance, in a distributed algorithm, this can be some name for the algorithm followed by an id of the component of the algorithm. In order to avoid confusion among the objects of different applications and programmers, one can either use different port numbers, or include in `object-name` a component that uniquely identifies the programmer.

Registering an object with the `rmiregistry` can be done with

[1] Please don't use hardcoded IP addresses for localhost. Just use "localhost".

```
java.rmi.Naming.bind(name, object);
```

where `name` is a name with the URL syntax defined above, and `object` is an object of a remote class.

Looking up a remote interface is done by

```
object = (typecast to object type) java.rmi.Naming.lookup(name);
```

where `object` is the reference to a remote object and `name` is the URL-name with which the remote object has registered itself. As a consequence, the local object has to know where the remote object resides, because the host name is a part of `name`, and so the `rmiregistry` is not really a system-wide naming service. If an object wishing to call a method of a remote object does not know where the remote object resides, it may have to consult the registries on a number of hosts to find this out.

Each user has to start his own registry (see Section 6).

The RMI Security Manager

In some cases, RMI invocations can be rejected when no so-called RMI Security Manager is installed. This can be done by including the following lines of code into your Java program:

```
// Create and install a security manager
   if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
   }
```

Note that, contrary to what one would expect, this might also break RMI calls.

If you are working on a single host, there is no need to use a security manager. However, if your client and server are on different machines, when the client downloads classes during runtime, the JVM needs to make sure that these classes are not hostile, and so you need to specify a security manager. RMI's class loader will not download any classes from remote locations if no security manager has been set.

2 Sending and Receiving Messages with Java/RMI

The only interaction between the processes of a distributed algorithm consists of sending and receiving messages. There exist languages for distributed systems that include primitives for doing this in a very explicit and natural way. However, although message passing can be implemented in an object-oriented language like Java, it has to be done in a rather artificial way by "abusing" the RMI mechanism. It is one of the subgoals of this lab to actually implement message passing in Java RMI and to apply this in the implementation of distributed algorithms. The way to do this is to simply have the sender of a message call a method supplied by the receiver with the message as a parameter.

3 Threads

When there are different tasks within a single program that have to run in parallel, it is useful to use threads, that is, different lines of execution. One way to create a thread is to declare a class that implements the `Runnable` interface, which is an interface that has a `run` method defined. In this method, all the work of the thread is performed. If `my-object` is the reference to an object of such a class, then a new thread for it can be created and started with

```
new Thread(my-object).start();
```

4 Critical Sections in Java

Different methods in an object may access the same data. If this happens without proper locking the (data in the) object, incorrect results may be obtained. In Java, method definitions accessing shared data should be preceded by the `synchronized` keyword, like in

```
public synchronized void method1() {
   ...
}
```

When a thread invokes a synchronized method, the object is locked, and so the method's code is in effect a critical section. Usually it is better only to synchronize the part of the method that is a critical section. Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the lock:

```
public void method1() {
   ...
   // Here starts the critical section
   synchronized(this) {
      ...
   }
   ...

}
```

Java 1.5 introduced the `java.util.concurrent` package for developing multithreaded applications. For synchronizing access to critical sections, classes in the `java.util.concurrent.locks` package can be used:

```
import java.util.concurrent.locks.ReentrantLock;

private ReentrantLock lock = new ReentrantLock();

lock.lock(); // blocks until current thread gets the lock
try{
    // critical section
}
finally{
lock.unlock();
}
```

See `http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/concurrent/locks/ReentrantLock.html` for the details.

5 The Structure of Programs for Distributed Algorithms

An appropriate structure for a Java program that implements a distributed algorithm consisting of multiple processes is to define interfaces and classes with the following functionality (here, `DA-name` is a user-chosen name):

1. The remote interface `DA-name_RMI` defines the methods that can be called remotely (see Section 1).

2. The remote algorithm class `DA-name` implements the remote interface `DA-name_RMI`, in which the actual work of a single process of the distributed algorithm is performed.

3. The main class `DA-name-main` creates all the processes of the distributed algorithm that will run on a single host. This can also be done using a script.

6 Program Invocation

Before a program implementing a distributed algorithm can really do its work, the registry has to be started on each of the hosts that will participate in the distributed algorithm. Starting a registry can be done in one of two ways, manually with a user command, or from a Java program. With the first option, it is convenient to have the registry run in the background, so the following command can be used:

```
rmiregistry &
```

With the second option, the code of the program of the distributed algorithm should use the class `LocateRegistry`, which has as one of its methods `createRegistry`, in the following way (see also Section 7):

```
public static void main(String args[]) {
   try {
      java.rmi.registry.LocateRegistry.createRegistry(1099);
   } catch (RemoteException e) {
      e.printStackTrace();
   }
}
```

7 Web Resources

The following links contain useful material on the use of Java/RMI:

- `http://java.sun.com/products/jdk/rmi`
- `http://java.sun.com/docs/books/tutorial/rmi`
- `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/`
- `http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`
- `http://java.sun.com/j2se/1.5.0/docs/api/java/rmi/registry/LocateRegistry.html`
- `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`
- `http://java.sun.com/developer/onlineTraining/rmi/RMI.html`
- `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`

8 Problems and Solutions

Below is a list of some problems that may be encountered during the lab, and their solutions.

1. **Problem:** RMI connection refused.

   Symptom:

   ```
   $ java MyClient
   java.rmi.server.ExportException: Port already in use: #portnumber#;
   nested exception is: java.net.BindException: Address already in
   use: JVM_Bind
   ```

   Solution: Kill running `java.exe` processes that hog the RMI registry ports. Also make sure that your program shuts down RMI registries upon closure.

2. **Problem:** RMI connection refused.

   Symptom:

   ```
   $ java MyClient
   java.rmi.ConnectException: Connection refused to host: 129.158.228.127
   ```

   Solution: Write a `my.policy` file:

   ```
   grant {
      permission java.security.AllPermission;
   };
   ```

   Please note that it grants AllPermission, and not AllPermissions (last 's' omitted). Also, this file must be placed in the root-path of the package. *The software supplied to the students already contains a* `my.policy` *file*.

   Launch the program with a user-specified permission:

   ```
   $ java -Djava.security.policy=my.policy MyClient
   ```

   When using Eclipse, please be aware to use to *run configuration* options to set the above permission as a VM argument.

3. **Problem:** RMI registry cannot find user classes.

   Solution: Make sure that the class path is set correctly not only for invoking the program itself, but also for starting `rmiregistry`. If you have the current directory ('.') in your class path, the easiest solution is to always start `rmiregistry` in the same directory where the classes are located that you plan to be doing `lookup()`.

4. **Problem:** Warning with serializable classes

   Symptom: Warning when compiling that no serialVersionUID is found.

   Solution: Add to the serializable classes:

   ```
   /* Not necessary to include in first version of the class, */
   /* but included here as a reminder of its importance. */
   private static final long serialVersionUID = 752647115562277614 7L;
   ```

   For a detailed description, see `http://www.javapractices.com/Topic45.cjp`.

9 Questions and Answers

Below is a list of some questions that may occur during the lab, and their answers.

1. **How does a client find a remote RMI service?**
   Using a naming or directory service which is the *rmiregistry* running on a well-known host and port number (by default on port 1099). The client can access the registry either using the `java.rmi.Naming` class or the `java.rmi.registry.LocateRegistry` class. After obtaining a reference to the registry it just calls the lookup method to retrieve a remote reference for the service.

2. **Why do I get the following exception**?

```
Caused by: java.lang.ClassNotFoundException: Test_Stub
        at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:252)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:320)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:247)
        at sun.rmi.server.LoaderHandler.loadClass(LoaderHandler.java:434)
        at sun.rmi.server.LoaderHandler.loadClass(LoaderHandler.java:165)
```

   The stub classes are not found. Make sure that you generate the stub classes with rmic and you start the server java process with **-Djava.rmi.server.codebase=file:** argument and specify the location of the stub classes. For example
   java -Djava.rmi.server.codebase=file:/home/myuser/classes/ MyMainClass.
   A codebase can be defined as a location from which classes are downloaded. The **java.rmi.server.codebase** property value specifies one or more URL locations from which stub classes (and any classes needed by the stubs) can be downloaded by the client from the server process.

3. **What are some rmiregisty command line arguments or JVM arguments that may help debugging?**
   For class loader issues you can try
   rmiregistry -J-Dsun.rmi.loader.logLevel=VERBOSE.
   Please check
   `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/sunrmiproperties.html` for other properties for Java 1.4 and
   `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/sunrmiproperties.html` for Java 1.5,

4. **Do I always need to specify a security policy file?**
   If you are working on a single host there is no need to specify one. However, if your client and server are on different machines when the client downloads the classes during runtime the JVM needs to make sure that downloaded classes are not hostile and so you need to specify one.

5. **What are the needed classes for a typical deployment where clients and the server are on different machines?**
   For the server the following classes must be on the classpath:

   - Remote service interface
   - Remote service implementation
   - Stubs
   - Other classes used by the server

   For the client the following classes must be on the classpath:

   - Remote service interface
   - Stubs
   - Server classes used by the client (e.g. return values)
   - Other classes used by the client

6. **Are remote methods always called by different threads by the RMI runtime?**
   No. The RMI specification specifies that a remote object implementation may or may not execute in a separate thread. Please check the spec for other details.