The Hello World example using Jeremie

April 12, 2002

This document describes step by step how the Hello World application has been developped with Jeremie. As you will discover, it is pretty much the same as writing a Java RMI application. Readers having never used RMI had certainly better go through the RMI tutorial. This example simply recalls a few things, and points out the few differences between RMI and Jeremie.

1 Step 1: write a remote interface

The first step when writing a Jeremie (or RMI) application is to describe the interfaces to be accessed remotely. The only thing to do is to make these interfaces extend `java.rmi.Remote`.

In the following, we'll use this remote interface specification[1] as an example:

```
25
26 import java.rmi.Remote;
27 import java.rmi.RemoteException;
```

Like in RMI, the interface must extend Remote.

```
30 public interface Hello extends Remote {
```

Like in RMI, the method must declare a RemoteException.

```
33    String sayHello() throws RemoteException;
34 }
```

2 Step 2: write a server

The file Server.java[2] contains an implementation of the interface `Hello` and a `main` method to run the server. The only difference with a similar example written using RMI is that the imported classes are different, and the naming service is not invoked exactly in the same way.

```
25
26 import java.rmi.RemoteException;
27 import org.objectweb.jeremie.libs.binding.moa.UnicastRemoteObject;
28 import org.objectweb.jeremie.libs.services.registry.Naming;
```

[1] contained in `examples/jeremie/helloWorld/srv/Hello.java`
[2] contained in `examples/jeremie/helloWorld/srv/Server.java`

Like in RMI, the easiest way to declare that an object may be accessed remotely is to make it extend a UnicastRemoteObject. Jeremie provides several implementations of UnicastRemoteObject. The implementation chosen here will multiplex all objects extending it on the same TCP/IP connection.

```
34 class HelloImpl extends UnicastRemoteObject implements Hello {
```

The constructor must declare the RemoteException.

```
38    HelloImpl() throws RemoteException {
39    }
```

Straigtforward implementation of the sayHello method.

```
42    public String sayHello() {
43       return "Hello World!";
44    }
45
46 }
```

The Server class simply contains a main method to start the server.

```
49 public class Server {
50    public static void main (String[] args) {
51       try {
52          String registryHost = "";
53          if (args.length != 0) {
54             registryHost = args[0];
55          }
```

This call registers a new Hello implementation in the JRMI registry. `registryhost` represents the machine on which the registry is currently running.

```
60          Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
61
62          System.out.println("Hello Server ready !");
63       } catch (Exception e) {
64          System.err.println("Hello Server exception");
65          e.printStackTrace();
66       }
67    }
68 }
```

3 Step 3: Compile the java source files and generate the stub code

Like in RMI, the stub compiler is invoked on the server class. It means in particular that the server must have been compiled before trying to generate the stubs. All these steps are performed automatically if you use the provided `Makefile`[3]: Simply type `make` or `make all` to compile everything.

[3] contained in `examples/jeremie/helloWorld/srv/Makefile`

4 Step 4 write a client

The file `Client.java`[4] contains a client for our server. The client only needs to have access to the `Hello` interface. Here again, the code is nearly identical to that of a client written for Java RMI.

```
25
26 import java.rmi.RMISecurityManager;
27 import org.objectweb.jeremie.libs.services.registry.Naming;
```

The Client class only contains a main method.

```
30 public class Client {
31    public static void main(String args[]) {
32       try {
```

It is necessary to set a security manager to let the client open new connections, download code, etc. A security policy file is provided to the client to grant it some rights. See the Makefile for details.

```
39          System.setSecurityManager(new RMISecurityManager());
40          String registryHost = "";
41          if (args.length != 0) {
42             registryHost = args[0];
43          }
```

This call retrieves a reference to the Hello object registered by the server.

```
48          Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
49          System.out.println();
50          System.out.println(obj.sayHello());
51
52       } catch (Exception e) {
53          System.err.println("Hello Client exception");
54          e.printStackTrace();
55       }
56    }
57 }
```

If you use the provided `Makefile`[5], you just need to type `make` or `make all` to compile your client.

5 Step 5: run your client and server

You are now ready to start the different applications.

- In the `srv` directory:
  - `make jrmiregistry` starts the name server;
  - `make server` starts the Hello World server;
- In the `clt` directory, `make client` starts the client.

[4] contained in `examples/jeremie/helloWorld/clt/Client.java`
[5] contained in `examples/jeremie/helloWorld/clt/Makefile`