Dynamic Hello World !

April 12, 2002

Jérémie offers the ability to define dynamic implementations of servers, i.e., implementations that need not directly implement the remote interfaces they offer, much like the Dynamic Skeleton Interface defined in the OMG CORBA specifications.

This document aims to describe how the Hello World application may be implemented by a dynamic implementation. The reader should first be familiar with the standard hello world implementation.

1 Step 1: write a remote interface

This first step is exactly the same as in the standard Hello World application. The first step 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;

31 public interface Hello extends Remote {
```

Like in RMI, the method must declare a RemoteException.

```
34    String sayHello() throws RemoteException;
35 }
```

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.

```
25
26 import java.rmi.RemoteException;
27 import org.objectweb.jonathan.apis.kernel.Kernel;
28 import org.objectweb.jonathan.apis.kernel.JonathanException;
29 import org.objectweb.jeremie.apis.binding.DynamicRemote;
30 import org.objectweb.jeremie.libs.binding.moa.UnicastRemoteObject;
31 import org.objectweb.jeremie.libs.services.registry.Naming;
```

[1] contained in `examples/jeremie/dynamicHello/srv/Hello.java`

[2] contained in `examples/jeremie/dynamicHello/srv/Server.java`

Dynamic implementations need only implement the DynamicRemote interface.

```
34 class DynamicHelloImpl implements DynamicRemote {
```

This method must return a class (or interface) implementing all the remote interfaces the dynamic implementation virtually implements.

```
39    public Class getImplementedClass() {
40       return Hello.class;
41    }
```

The `invoke` method is used to actually invoke a method. In this example, we do not check whether the operation name and types correspond to an actually defined method... This check should of course be performed in real-life code !

```
48    public Object invoke(String op_name,
49                         Class[] param_types,
50                         Object[] parameters) throws RemoteException {
51       return "Hello World!";
52    }
53 }
```

Like in the standard Hello world example, the Server class simply contains a main method to start the server.

```
57 public class Server {
58    public static void main (String[] args) {
59       try {
60          String registryHost = "";
61          if (args.length != 0) {
62             registryHost = args[0];
63          }
```

Dynamic implementations must be exported using the static `exportObject` method. Nothing else is specific.

```
67          DynamicHelloImpl impl = new DynamicHelloImpl();
68          UnicastRemoteObject.exportObject(impl);
```

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

```
73          Naming.rebind("jrmi://" + registryHost + "/dynhelloobj", impl);
74
75          System.out.println("Dynamic Hello Server ready");
```

This is just to check that local calls may be performed properly.

```
78          Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost +
79                                            "/dynhelloobj");
80          System.out.println();
81          System.out.println(obj.sayHello());
82
```

```
83       } catch (Exception e) {
84          System.err.println("Dynamic Hello Server exception");
85          e.printStackTrace();
86       }
87    }
88 }
```

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

When, like in our example, only one interface is implemented by the server, the stub compiler may be invoked on its class. Otherwise, it would be necessary to create a new interface extending all the remote interfaces implemented by the dynamic implementation (or an abstract class implementing these interfaces) and compile it before generating the required stubs. Note that with dynamic implementations, only standard (non optimised) stubs may be used.

These steps are performed automatically if you use the provided `Makefile`[3]: Simply type `make` or `make all` to compile everything.

4 Step 4 write a client

The client code is identical to that of the stanard Hello Worls example, except the name used in the registry.

```
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.

```
47          Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/dynhelloobj"
48          System.out.println();
49          System.out.println(obj.sayHello());
50
51       } catch (Exception e) {
```

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

```
52          System.err.println("Hello Client exception");
53          e.printStackTrace();
54       }
55    }
56 }
```

If you use the provided `Makefile`[4], you just need to type `make` or `make all` to compile your client.

5 Step 5: run your client and server

Here again, the steps to follow are identical as in the standard case:

- 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/dynamicHello/clt/Makefile`