The Hello World example using the David RMI/IIOP implementation

April 12, 2002

David provides an RMI/IIOP implementation. Contrary to Jeremie, which is not a standard implementation of RMI, the provided implementation strictly conforms to the standard. This document describes step by step how the Hello World application has been developped with David's RMI/IIOP. However, it is not a tutorial on distributed programming with Java or RMI/IIOP. This example simply describes the main steps, and outlines some David specificities.

1 Step 1: write a remote interface

The first step when writing an RMI/IIOP 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:

```
23
24 package hello;
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. Since David doesn't provide yet a JNDI interface to the David naming service, the way used to provide the server reference is quite specific.

[1] contained in `examples/david/helloRMI/srv/Hello.java`

[2] contained in `examples/david/helloRMI/srv/Server.java`

```
23
24 package hello;
25
26 import java.rmi.RemoteException;
27 import javax.rmi.PortableRemoteObject;
28 import javax.rmi.CORBA.Util;
29
30 import org.omg.CORBA.ORB;
31 import org.objectweb.david.libs.helpers.IORHelpers;
```

In the same way as in RMI, the easiest way to declare that an object may be accessed remotely is to make it extend PortableRemoteObject.

```
35 public class Server extends PortableRemoteObject implements Hello {
```

The constructor must declare the RemoteException.

```
39    public Server () throws RemoteException {}
```

Straigtforward implementation of the sayHello method.

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

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

```
47    public static void main (String[] args) {
48       try {
49          Server obj = new Server();
```

The expected way to give access to the reference of the exported object would be to register it in a JNDI naming service. For now, the David naming service has not been provided with a JNDI interface. We thus use a less standard method: we first retrieve a reference to an ORB (it exists when RMI/IIOP is used, but usually doesn't need to be explicitly manipulated), then the object reference is turned into a a string that is written to a file.

```
58          ORB orb = ORB.init(args,null);
59          String ior = orb.object_to_string(Util.getTie(obj).thisObject());
60          IORHelpers.writeIORToFile(ior,args[0]);
```

The following line simply states that the server is ready to receive incoming requests.

```
64          System.out.println("Hello Server ready");
65
66       } catch (Exception e) {
67          System.out.println ("HelloServer Exception: " + e.getMessage());
68          e.printStackTrace ();
69       }
70    }
71
72 }
```

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.

Please note that David provides its own stub compiler, which is used if you use the provide Makefile. However, it would be perfectly possible to use another stub compiler like `rmic -iiop`, provided with JDK 1.3.

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.

```
23
24 package hello;
25
26 import java.rmi.RemoteException;
27 import javax.rmi.PortableRemoteObject;
28 import java.rmi.RMISecurityManager;
29
30 import org.omg.CORBA.ORB;
31 import org.objectweb.david.libs.helpers.IORHelpers;
```

The Client class only contains a main method.

```
34 public class Client {
35    public static void main (String[] args) {
36       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.

```
43          System.setSecurityManager(new RMISecurityManager());
```

The way to retrieve a reference to the server implementation is dual to that used on the server side. The stringified object reference is first read from a file, then turned into a CORBA object.

```
48          String ior = IORHelpers.readIORFromFile(args[0]);
49          ORB orb = ORB.init(args,null);
50          org.omg.CORBA.Object obj_ref = orb.string_to_object(ior);
```

The obtained CORBA object reference must then be narrowed, using the `narrow` method of `PortableRemoteObject`. Note that a direct cast can't be used here.

[3] contained in `examples/david/helloRMI/srv/Makefile`

[4] contained in `examples/david/helloRMI/clt/Client.java`

```
55          Hello hello = (Hello)
56             PortableRemoteObject.narrow (obj_ref,Hello.class);
```

Now, the call to the server:

```
59          System.out.println(hello.sayHello());
60       } catch (Exception e) {
61          e.printStackTrace ();
62       }
63    }
64 }
```

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 server` starts the Hello World server;
- In the `clt` directory, `make client` starts the client.

6 David specific options

If you look at the provided `Makefiles`, you'll see that the java interpreter is started using numerous options. Some of them are compulsory, others are only optional.

6.1 Options to use the David ORB

When using David, you must use the David ORB. You must thus start the java interpreter with the following options, telling to the interpreter which ORB implementation to use:

```
-Dorg.omg.CORBA.ORBClass= \
   org.objectweb.david.libs.binding.orbs.iiop.IIOPORB
-Dorg.omg.CORBA.ORBSingletonClass= \
   org.objectweb.david.libs.binding.orbs.ORBSingletonClass
```

6.2 Options to use the David RMI/IIOP implementation

To use the David RMI/IIOP implementation, you must specify the names of delegation classes to be used by the virtual machine, using the following options:

```
-Djavax.rmi.CORBA.StubClass= \
   org.objectweb.david.libs.stub_factories.rmi.StubDelegate \
-Djavax.rmi.CORBA.PortableRemoteObjectClass= \
   org.objectweb.david.libs.binding.rmi.ORBPortableRemoteObjectDelegate \
-Djavax.rmi.CORBA.UtilClass= \
   org.objectweb.david.libs.helpers.RMIUtilDelegate
```

[5] contained in `examples/david/helloRMI/clt/Makefile`

6.3 Setting the ValueHandler implementation

David doesn't provide a standard implementation of the `ValueHandler` interface, because such an implementation depends on the virtual machine used. SUN provides an implementation of the interface with the JDK 1.3; to use it, the following option should be used:

```
-Ddavid.rmi.ValueHandlerClass=\
   com.sun.corba.se.internal.io.ValueHandlerImpl
```

If no implementation is available, you can still use David, but Java serialization will be used instead of CORBA serialization. In this case, you won't be able to interoperate with other RMI/IIOP implementations.

6.4 Other options

- Using Java serialization instead of CORBA serialization

  Even if a `ValueHandler` implementation is available, Java serialization may be used instead of CORBA serialization. This may be useful, either for performance reasons, or to overcome some bugs of SUN's value handler implementation. To do so, the following option must be used:

  ```
  -Ddavid.iiop.use_java_serialization=true
  ```

  This option is needed both on the server and client sides. On the server side, it is needed so that the server reference contains an indication that the server is ready to accept requests encoded using Java serialization. On the client side, it is needed to decide whether Java serialization should be used or not when interacting with properly exported servers.

  Note that using that option doesn't prevent interoperability. Note also that this option is useless when no `ValueHandler` implementation is provided.

- Avoiding parameters and result copies

  The RMI/IIOP standard stipulates that, when a client and a server are executed in the same virtual machine, parameters and results should anyway be copied to preserve the remote call semantics.

  These copy operations may be very costly, and are usually not needed by applications. To avoid them, the following option should be used:

  ```
  -Ddavid.rmi.local_copy=false
  ```