The Hello World example using David

April 12, 2002

This document describes step by step how the Hello World application has been developped with Jonathan. However, it cannot be considered as a CORBA tutorial, and readers having no familiarity at all with CORBA are strongly encouraged to refer to some CORBA tutorial.

1 Step 1: write an IDL specification

The first step when writing a CORBA application is to describe the interfaces to be accessed remotely using the CORBA *Interface Definition Language* (IDL). In the following, we'll use this IDL specification[1] as an example:

```
1
2 interface Hello {
3    string sayHello();
4 };
```

2 Step 2: compile your IDL specification

The IDL specification is used to generate the code of the objects responsible for transmitting requests to remote entities. The generation is carried out by an *IDL compiler*. The Jonathan distribution provides such a compiler, written in Java. The following command starts the compilation of `Hello.idl`:

```
java org.objectweb.david.tools.idlcompiler.Idl2Java -p idl test.idl
```

Calling the compiler this way assumes that the Jonathan classes are in your classpath. This operation is performed automatically if you use the provided `Makefile`[2].

The compilation results in Java source files. In our example, these files are generated in the directory `idl`. These files have been generated below the `ti` package because the `-p` option of the IDL compiler has been used. For more information about the IDL compiler, have a look at its HTML documentation.

3 Step 3: write servers

The file Server.java[3] contains implementations for the interface `Hello` specified in `Hello.idl`, and a `main` method to run the server. Let's go through the code:

[1] contained in `examples/david/helloWorld/Hello.idl`
[2] contained in `examples/david/helloWorld/Makefile`
[3] contained in `examples/david/helloWorld/Server.java`

```
25
26 import org.omg.CORBA.ORB;
27 import org.omg.CosNaming.NamingContext;
28 import org.omg.CosNaming.NamingContextHelper;
29 import org.omg.CosNaming.NameComponent;
30 import idl.*;
```

`HelloImpl` is the class implementing the `Hello` interface. It *extends* `_HelloImplBase`: `_HelloImplBase` is the class of skeletons generated from the `Hello` interface IDL specification. It encapsulates the mechanisms necessary to manipulate `Hello` references in a CORBA distributed context (in particular the `org.omg.CORBA.Object` methods). Extending a skeleton is the simplest method to tell the system that an interface may be used in a remote invocation.

```
41 class HelloImpl extends _HelloImplBase {
42
43    HelloImpl() {}
```

The following code implements in a straightforward way the `sayHello` method defined in interface `Account`.

```
47    public String sayHello() {
48       return "Hello World!";
49    }
50
51 }
```

The `Server` class is the main class. It just contains a `main` method.

```
55 public class Server {
56    public static void main (String[] args) {
57       try {
```

The following line is the standard way to initialize an ORB. The first argument contains the arguments passed to the main operation, the second one is a `Properties` instance. These methods are used to initialize the ORB. One property is particularly important:

`org.omg.CORBA.ORBClass`

This property indicates which ORB implementation (and in particular, which protocol) should be used. In our case, this property is not set using the second argument, but through an additional argument (`-Dorg.omg.CORBA.ORBClass=...`) passed to the java virtual machine when starting the server (see the Makefile for details).

```
71          ORB orb = ORB.init(args,null);
```

This simply creates an implementation for Hello.

```
74          Hello hello = new HelloImpl();
```

The next line “connects” the instance to an ORB. The role of this operation is to register the newly created instance in the ORB so that it can be used as a parameter in remote invocations, exported to a name server, or receive invocations from remote objects.

The skeletons generated by the David IDL to Java compiler are protocol independent: they may be used in invocations using the standard IIOP protocol, or a multimedia stream specific protocol, or whatever protocol is accessible. The protocol chosen depends on the ORB instance.

Note that a given implementation may be exported to several different ORBs, if available.

```
91          orb.connect(hello);
```

David provides a standard COS name server. A name server is a server application containing an association table between names and running servers, and allowing remote objects to register new interfaces under a given name, or retrieve registered interfaces by providing their name. The next invocation retrieves a reference to the running default name server: such a name server should be running when this code is executed (to start a name server see the Makefile). A reference to the name server may be retrieved using the initial references mechanism of CORBA. The first invocation on `orb` returns a CORBA object representing the name server; This object reference must then be narrowed to obtain a Java object of the appropriate type.

```
108         org.omg.CORBA.Object ns_ref =
109            orb.resolve_initial_references("NameService");
110         NamingContext ns = NamingContextHelper.narrow(ns_ref);
```

The following line shows how the server is registered in the name server. The first argument is the ”name” given to the server.

```
114         ns.rebind(new NameComponent[] { new NameComponent("helloobj","") },hello);
```

This call blocks the calling thread until the `shutdown` method is called on `orb` (in our case, never!).

```
118         System.out.println("Hello Server ready");
119         orb.run();
120       } catch (Exception e) {
121         System.err.println("Hello Server exception");
122         e.printStackTrace();
123       }
124    }
125 }
```

4 Step 4 write a client

The file `Client.java`[4] contains a client for our server. Here is the code:

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

```
25
26 import org.omg.CORBA.ORB;
27 import org.omg.CosNaming.NamingContext;
28 import org.omg.CosNaming.NamingContextHelper;
29 import org.omg.CosNaming.NameComponent;
30 import idl.*;
```

The Client code only consists in a `main` method.

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

The ORB initialization, and the retrieval of the name server reference are preformed exactly like in the Server case.

```
39          ORB orb = ORB.init(args,null);
40          org.omg.CORBA.Object ns_ref =
41             orb.resolve_initial_references("NameService");
42
43           NamingContext ns = NamingContextHelper.narrow(ns_ref);
```

After having retrieved a reference to the name server, the client retrieves a reference to a server registered in the name server. Like for the name server in the Server code, the obtained reference must be narrowed (and not simply cast).

```
49          org.omg.CORBA.Object obj_ref =
50             ns.resolve(new NameComponent[] { new NameComponent("helloobj","") });
51          Hello obj = HelloHelper.narrow(obj_ref);
52
53          System.out.println(obj.sayHello());
54
55       } catch (Exception e) {
56          System.err.println("Hello Client exception");
57          e.printStackTrace();
58       }
59    }
60 }
```

5 Step 5: run your client and server

The provided `Makefile` automates all the compilation phases. To compile the IDL file and the Java files, simply type `make` at a shell prompt in the example directory.

Once compiled, you may start the different applications:

- `make cosnaming` starts the name server;
- `make server` starts the Hello World server;
- `make client` starts the client.