Implementation of a distributed auction application using Java RMI and Java RMI-IIOP.

Abstract

This report discusses about the lab assignment 1 that consists on developing a distributed application employing Java RMI and Java RMI-IIOP technologies. The aim is to implement a program that manages an auction system. This report is mainly focused on the problems found as well as on conceptual conclusions. First of all, Java RMI development is explained and afterwards Java RMI-IIOP is discussed.

1. Development with Java RMI

Thanks to Java RMI, the distributed application implementation is not much more difficult than the centralized case. The realization process employing Java RMI is not too different from the centralized case because the whole communication part is hidden. The only thing that needs to be done by the developer is to register the remote object within the RMI register. In order to achieve that, it is necessary to create the remote object that provides the service and export it to the Java RMI runtime. This is done with the following code:

```
Server obj = new Server();
IItemManager stub = (IItemManager) UnicastRemoteObject.exportObject(obj, 0);
```

First of all, a server object is instanced, and then, the method `UnicastRemoteObject.exportObject` exports the remote object in order to be able to receive incoming remote method invocations. That `exportObject` method returns the stub that has to be passed to the clients. The returned stub implements the same set of remote interfaces as the remote object's class and contains the host name and port over which the remote object can be contacted.

Now, the stub must be registered within the Java RMI registry. A Java RMI registry is a simplified name service that allows clients to get a reference (a stub) to a remote object. In order to do that, the following code is used:

```
Registry registry = LocateRegistry.getRegistry(host, port);
registry.rebind("IItemManager", stub);
```

So now, the remote object stub is ready to be accessed by the client.

On the other hand, the client must implement this code for obtaining the stub:

```
Registry registry = LocateRegistry.getRegistry(host, port);
IItemManager stub = (IItemManager) registry.lookup("IItemManager");
```

The client first obtains the stub for the registry by invoking the static `LocateRegistry.getRegistry` method. Next, the client invokes the remote method `lookup` on the registry stub to obtain the stub for the remote object from the server's registry. Now, the client can invoke the methods on the remote object´s stub.

Finally, in conclusion, the most significant difference between the centralized case and the distributed one remains in the use of stubs and skeletons. On the one hand, in the centralized case, client object and server object are at the same machine, and they can interact directly by means of method invocations. On the other hand, in the distribute case using Java RMI, client object and server object could run in different machines. On the client side we can find the client object and the stub, which is used by the client object as a reference of the server object. On the server side, we can find the server object and the skeleton that forwards the remote invocations from the client to the server object itself.

DISTRIBUCION CON OTROS GRUPOS.

2. Development with Java RMI-IIOP.

The differences with respect to the centralized case are nearly the same to RMI case. Likewise the programming process using Java RMI-IIOP is very similar to Java RMI. It is also necessary to extend to Remote interface. Although, there are some differences that will be discussed now.

First of all, the interfaces implementations extend to `PortableRemoteObject`. Moreover, the constructor of `javax.rmi.PortableRemoteObject` must be invoked, in order to export the remote object. This occurs even if there is not an explicit invocation.

When implementing the server code, you must instant the servant (same as using Java RMI) and the, you must publish the reference in the Naming Service using JNDI API.

```
ItemManager obj = new ItemManager();
Context initialNamingContext = new InitialContext();
initialNamingContext.rebind("IItemManager", obj);
```

This is the most important difference with Java RMI, because CORBA Naming Service is employed instead of RMI Registry.

In the client code, it is necessary to get the object reference from the Name Service using a JNDI call.

```
objref = ic.lookup("IITemManager");
```

Then, the object reference must be narrowed to the concrete type in order to invoke the methods.

```
stub = (IItemManager) PortableRemoteObject.narrow(objref, IItemManager.class);
```

To summarize, the changes with respect to RMI are the remote interface implementation and the use of CORBA Name Service. In addition, the rmic compiler is run with the -iiop option to create CORBA-compatible stub and skeleton files:

- _IItemManager_Stub.class - the client stub
- _ItemManager_Tie.class - the server skeleton

Comparing the architectures with Java RMI, it can be observed that it is very similar. The skeleton is the server part and the stub is in the client part. However the protocol used to communicate the remote reference layer is IIOP instead of JRMP.

The main advantage of Java RMI-IIOP is that the interoperability with CORBA object is possible, whatever the programming language employed. In Java RMI is only possible the communication between Java objects. This assignment does not show it because both client and server are developed using Java.

Cosas que tiene que conocer el cliente:

- El codebase: no se puede acceder a un directorio de otra cuenta
- El nombre del servicio debe ser conocido por el cliente
- El puerto y el servidor
- La interfaz. (IItemManager)