15-440 Distributed Systems

Lab 02

Michael Wang - mhw1
William Maynes - wmaynes

October 11, 2013

1 General Design

Our RMI facility is split into two packages: rmi.com, which handles communications between the client and the server, and rmi.reg, which handles the registry. The rmi.com package may be further split into two distinct sections: the classes which handle the Remote Object Reference (ROR) table and the RMI server, and the client-side marshaller.

1.1 Remote Objects

We require several things from any client who wishes to implement a remote object. First of all, the remote object must implement the Remote440 interface, solely as a flag to indicate that a class can have methods invoked remotely. Secondly, certain guarantees about the remote object must be met: any access to or modification of fields within the object must be through methods. Every function of any object that implements Remote440 must be declared to throw rmi.com.RemoteException or one of its supertypes.

Additionally, every method parameter must either implement Serializable or be a stub for a remote object itself. In this way, we can avoid difficulties in attempting to directly set or get values of fields from remote objects, and the requirement on the method parameters ensures that marshalling when we attempt to make the function call will succeed.

Furthermore, no method parameter can be of a primitive type. For example, any parameters of type `int` should be replaced with parameters of type `Integer`. This is to avoid complications involving serialization. Note that, due to the way Java handles wrapper classes for primitives, it is possible to directly alter the function signature of any affected functions without altering function behavior.

Side-effects should be limited to parameters that implement the Remote440 interface. Due to Java's pass-by-value semantics during function calls and the way in which we marshal function invocation requests, supporting side-effects for any objects that are not remote objects is prohibitively complicated.

Finally, each remote object must have a stub class that contains a ROR for its parent object, and which handles all remote method invocation requests by overriding every method in the parent class.

Note that, due to the limitations of Java's pass-by-value semantics, side-effects in function calls are only supported for objects that implement the Remote440 interface.

1.2 Stubs

Stub objects must both extend the parent Remote440 object and implement the Stub interface. While no stub compiler has been provided, implementation of such a compiler is certainly possible. Constructors of the parent object can be given arbitrary values, and all parent methods should be overridden using calls to `RMIInvocation.invoke` (described in the Client-Side Communications section below). Additionally, the Stub should store the ROR corresponding to its parent and, in its constructor, should accept said ROR. A sample Stub object, demonstrating how a stub class can be procedurally generated from a class that implements Remote440 has been provided for your reference in rmi.test.

1.3 Registry

The registry listens to a port specified on the command line. It accepts requests to bind a name to a ROR, unbind a name from a ROR, lookup a ROR given a name, and rebind a bound name to a different ROR. The registry can be instantiated on any host by using the `createRegistry` method provided in the `LocateRegistry` class, and providing the port number to use.

`LocateRegistry` also contains a method, `hasRegistry`, which polls a host at a specified port and attempts to find a registry there. While requests to the registry must utilize `RegistryRequest` objects, all such requests are handled by the framework, and at no point should the programmer need to make a request to the registry.

1.4 Client-Side Communications

On the client side, once the user has successfully looked up a remote object and retrieved the corresponding ROR, he can use call the `localize` function provided by the ROR to generate a Stub object that can be used to remotely invoke method calls. As described in the Stubs section above, the Stub object uses `RMIInvocation.invoke` make the remote message call.

`RMIInvocation.invoke` takes the name of the function being invoked, the type of the object on which the method is to be invoked, the ROR corresponding to the object, and the parameters for the method passed into the Stub object. All function parameters that implement Stub are converted into their corresponding RORs, and then `RMIInvocation.invoke` marshalls all the information passed into it, sending it to the server specified by the ROR.

Then, it listens for a response from the server (the process of which is described below), and returns a value or throws an exception as appropriate.

1.5 RMI Server

The RMI server can be run by creating an instance of `ServerTask` and running it using a `Thread` object. It contains a `RORTable`, mapping RORs to the objects they represent.

The server listens on a specified port for incoming RMI invocation requests. Once such a request is received, it unpacks the request and retrieves the corresponding object from the RORTable. Then, the server uses reflections to generate a `Method` object corresponding to the method being invoked. Then, for each ROR in the list of parameters given in the request, it converts that ROR into the corresponding Stub object, and invokes the method with the modified list of parameters.

If, at any point during this process, an exception is thrown, then the server returns a response containing an exception to the client. Otherwise, if the function returns a value, then a value is returned to the client. If the function has return type `void`, the return type is indicated, `null` is returned instead.

2 Compilation and Testing

2.1 Compilation

When in the same directory as `build.xml`, run `ant` to build the project.

2.2 Testing

Test files and example main files are provided in the `rmi.test` package: run them using `java -cp bin path.to.MainClass`. `RegMain` runs the registry, `TestMain` runs the RMI server, and `TestMain2` executes methods using the RMI server.. Alternatively, you can run `rmi.reg.Registry` on the command line using `java -cp bin rmi.reg.Registry <port number>`.