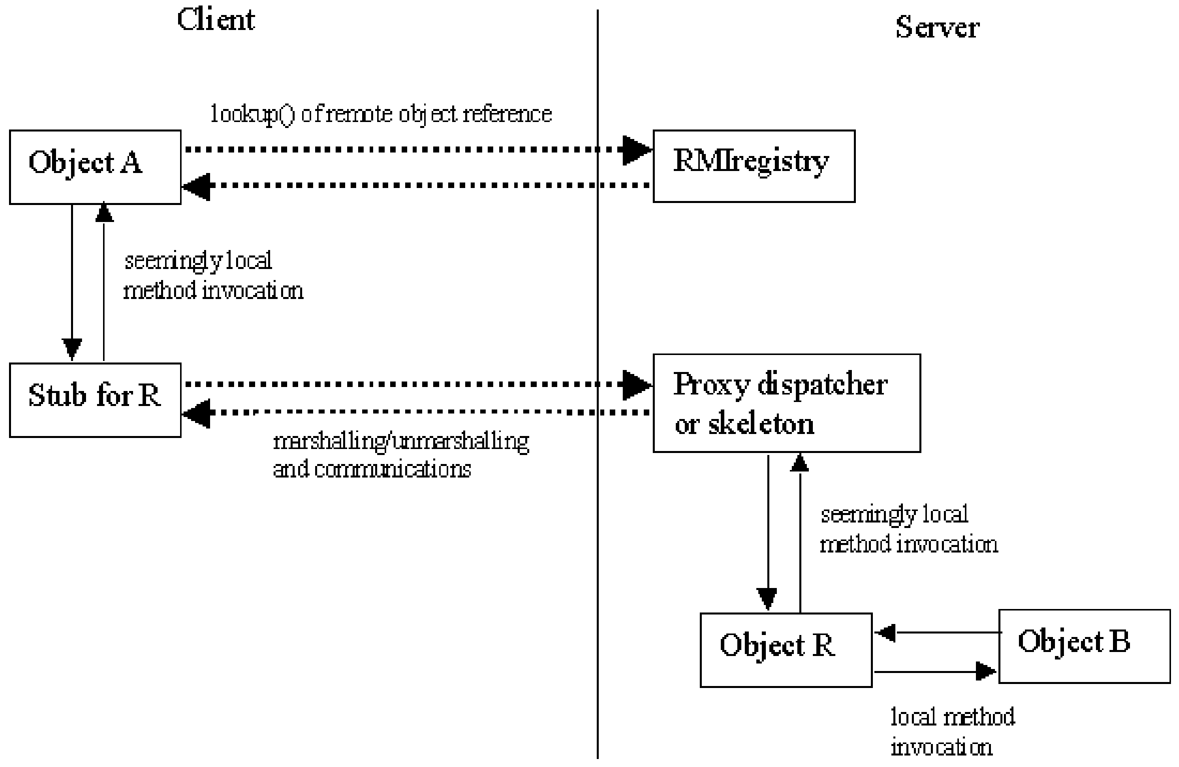
15-440 Project 2

Marshalling and Communication - RMI Version Implementation

Shri Karthikeyan
(skarthik)
Nandini Ramakrishnan
(nramakri)

Table of Contents

1. Project description


Client
Server
lookup() of remote object reference
Object A
RMIregistry
seemingly local method invocation
Stub for R
Proxy dispatcher or skeleton
marshalling/unmarshalling and communications
seemingly local method invocation
Object R
Object B
local method invocation

- The goal of this project is to write an implementation of Java's RMI i.e. giving the user the ability to call methods on remote objects, without making use of the java.rmi package.
- The remote objects reside on the server and the client wishes to run methods on these objects, even though they don't reside on the same host.
- This is achieved using remote object references, which are passed down from the server to the client. This lets the client know where the object resides and enables the client to create a proxy object, known as a stub, to invoke methods on this remote object.
- Once the stub has been created, it marshals the method invocation to a server-side object called skeleton. This handles the unmarshaling of the method invocation and can now make a local call on the source object.
- The return value is sent back to the client, while giving the client the impression that it just made a local call, when the method was actually executed on another machine.

2. Description of Components

Our project contains three packages - Server, Client and Utilities
In this section, we will describe the classes in these packages which have common functionality required for any remote object (eg. RemoteObjRef, RMI, Client etc.)
In the next section, we will focus on the test classes present in these packages (eg. Fibonacci, FibonacciInterface etc.)

a. Server:
   I. RMI: The server class, whose main function is continuously executing while listening for communication requests from clients. This class also launches the registry to which all server-side remote objects are binded.

   II. RMITransactions: Does the client-specific work of the server ie. looking up remote object references on the registry and creating skeletons for the client stub to communicate with.

   III. Registry: It defines a ConcurrentHashMap (for thread safety) which maps remote objects to the identifer strings that they were binded with by the server.

Example-specific classes on the server:

   IV. Fibonacci (implements FibonacciInterface): This class computes the all the fibonacci numbers up to n and returns an array list of them

   V. Fibonacci_skeleton: Acts as a proxy for the server. Unmarshals the method invocation and calls the method

   VI. ReverseString: This object has a method that reverses a string for you

   VII. ReverseString_skeleton: Acts as a proxy for the server. Unmarshals the method invocation and calls the method

b. Client:
   I. Client: The client parses the command line arguments, requests a lookup based on the service name and receives the return value from the stub created from the remote object reference. Client.java has specifically been written for the objects Fibonacci (service name=”fib”) and ReverseString (service name=”rev”).

Example-specific classes on the client:

II. Fibonacci_stub (implements FibonacciInterface): Proxy of Client which marshals the method invocation on Fibonacci

III. ReverseString_stub (implements ReverseInterface): Proxy of Client which marshals the method invocation on ReverseString

c. Utilities

I. RMIMessage: Contains constructors for all types of messages (eg. remote object reference, method invocations, return values, etc.) which are being sent either between the client and the server or the stub and the skeleton. Any communication between these modules will be wrapped in an RMIMessage Object.

II. RMIMessageDelivery: This is the standard communications module that handles serialization. It can either take in a host-and-port pair or a socket.

III. RemoteObjRef: Defines the remote object reference passed on from the client and the server. Takes in the IP Address, port number, interface name and service name.

IV. Remote440: Any remote interface will extend Remote440, which plays the role of java.rmi.Remote

----

Example-specific classes on utilities:

V. FibonacciInterface (extends Remote440): This is the interface the defines Fibonacci.

VI. ReverseInterface (extends Remote440): This is the interface the defines ReverseString.

3. Project Design

We developed two Remote objects(Fibonacci, ReverseString) residing on the server-side, and a client who can request method invocations on either of these objects.

To explain our design, lets take the example of the remote object, Fibonacci:

- The client requests a lookup of the service name “fib”, to see if it resides on the server. The service-name is wrapped in an RMIMessage object (which is the standard messaging format for any communication) and using an RMIMessageDelivery object (which is the standard communication module), we send it to the server.
    - *In our design, any object being sent between the client and the server (or their proxies) is wrapped in an RMIMessage object, and the serialization is done by an RMIMessageDelivery object.*
- The server, on loading, initialized the registry and performs the lookup method, which returns a RemoteObjRef object. The client receives the remote object reference, passes the host name of the registry to its localise() method and runs it.
    - *In our design, we are making the assumption that the registry resides on the same host as the server.*
- The localise() method creates a client stub, which is responsible for invoking the method on the remote object from the client side. The stub marshals the method invocation to the skeleton. The skeleton unmarshals the method invocation and uses reflection to make a call to the method on the local object (in the case of Fibonacci, we will be calling getFibonacciSeries(size). The result gets evaluated and the skeleton then marshals the result to the stub, which returns the unmarshalled value to the client.
    - *In our design, we have made use of stubs and skeletons to act as proxies of the client and the server respectively.*

Other design decisions:-

- The server binds the remote objects to the registry before it opens a ServerSocket to listen for incoming connections

- We implemented multi-threading so that our RMI Server is capable of handling multiple client requests

- We assume there is one operational RMI Server while executing our application and the RMI Registry resides on the same host.

4. How to execute the RMI Application

Directory path:

- `skarthik-nramakri`
    - `src`
        - `rmilab`
            - `server`
            - `client`
            - `utilities`
    - `Project2Report.pdf`

You can download the project folder onto the two nodes you will be testing on. On the node which will act as your server, compile the Java files in `rmilab.server` and `rmilab.utilities`. On the node which will act as your client, compile the Java files in `rmilab.client` and `rmilab.utilities`. ***For any compilation or running programs, you need to be at the root of the `src` folder.*** The steps are given below:

For the server node:
When you are in the src folder, type

NOTE: VERY IMPORTANT TO COMPILE UTILITIES FIRST THEN SERVER!!!!

```
> javac -Xlint:unchecked rmilab/utilities/*.java
> javac -Xlint:unchecked rmilab/server/*.java
```

Note: During compilations, you might get warnings (not errors), which have to do with our reflection methods. They do not affect the functioning of this program

To run the server, type

```
> java rmilab.server.RMI
```

For the client node:
When you are in the src folder, type

NOTE: VERY IMPORTANT TO COMPILE UTILITIES FIRST THEN CLIENT!!!!

```
> javac -Xlint:unchecked rmilab/utilities/*.java
> javac -Xlint:unchecked rmilab/client/*.java
```

To run the client, type

```
> java rmilab.client.Client
```

Once client and server are BOTH running, you will see a prompt to enter your request at the client node. Type your request in the following format,

```
> <object-name> <registry-host> <registry-port> <service-name>
  <method-name> <args>
```

Registry port is hardcoded to 1234
Below are sample commands that you can copy and paste to test out.

eg. if you want to view a Fibonacci series with 25 elements, type

```
> Fibonacci <registry-host> 1234 fib getFibonacciSeries 25
```

where `<registry-host>` is the same as the server host name. For your convenience, we print the server's host name at the server node, from where you can copy it down for testing

eg. If you want to view the reversed form of "RMIisCool" type

```
> ReverseString <registry-host> 1234 rev reverseString RMIisCool
```