15-640 Distributed Systems

Lab2 - RMI Facility in Java

-Amey Ghadigaonkar

6/24/2014

**15 - 640 Distributed Systems**

**Lab 2 – RMI Facility**

**-Amey Ghadigaonkar.**

**My RMI Design:**

I designed my RMI framework with an aim to simplify the implementation as much as possible for the Application Programmer. Thus, I don’t let the user mess with the received RoRs. I abstract away this handling like the actual RMI does by encapsulating this RoR inside a “Stub” object.

However, I tweaked the framework as described later in the report so that the user can also explicitly pass RoRs as parameters if he gets his hands on a RoR (I have a method getRoR() that returns the RoR inside the stub if the user wants to do this).

My framework can:

1. Ability to lookup existing remote objects. The bindnames are assigned at the server and the client does not have the power to name/rename remote objects.
2. Invoke methods on remote objects and pass/return remote/local objects.
3. A stub compiler to create simple stubs that are compatible with my framework

Things I did not have time to implement:

1. Transferring “.class“ files. However, I had a simple mechanism in mind to transfer the “.class“ file using sockets (the server will have a listening port to transfer these class files).
2. Distributed garbage collector. However, I read about this on the official Java docs (where I read about all the other stuff, too). I have a rough idea about it and would have loved to do it. But, shortage of time.

**Design Decisions and tradeoffs:**

I implemented this framework by referring primarily to the existing Java RMI framework. Like it, I also pass the Remote Object References implicitly without the knowledge of the Application Programmer. It is achieved by using the concept of “stub” as described below in point 7. The major decisions I took while developing the framework have been specified below:

1. Having a generic Remote440 interface:

This Remote440 interface is analogous to java.rmi.Remote class. However, my Remote440 class is just a wrapper for the objects to be handled by my framework. It does not specify any business logic or any methods.

1. Having a generic RemoteStub class:

This class is the pseudo object I will store at the client. This contains a generic method “invoke” which handles all the communication with the server and marshals and unmarshals arguments and gets return values. This stub is needed to create a generic stub which is not concerned with implementation of the method. It only concerns itself with abstraction of the implementation from the application programmer.

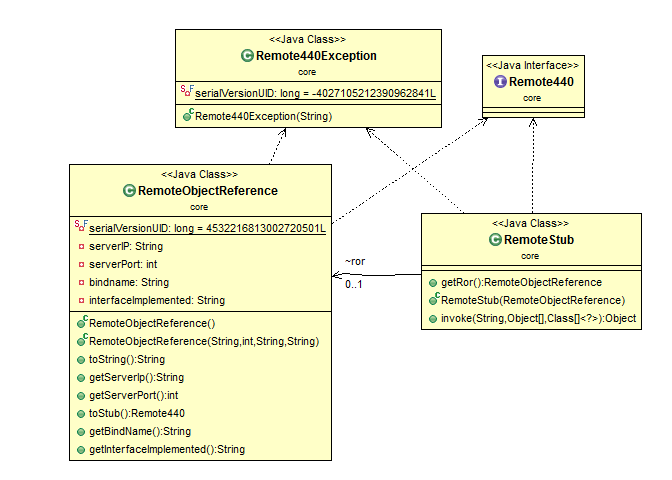


Figure 1 Remote440 class and its children

1. Having a generic Remote440Exception: This is a common exception that the server throws in order to hide the things that went wrong at the server or during communication. In my view, these errors are implementation level details that the user does not need to be aware of.
2. Having a standard Remote Object Reference: The RoR is an important part of the implementation that needs the required information to identify and communicate with the server which is holding the remote object. These RoRs are generated only at the server and thus can be used at only the server since the information required to use them is accessible only there.
3. Having a common Message class: One of the main lessons from the last assignment was that the network communication needs to be simplified to a point that I know exactly what is being transferred over the network at any point in time and that the contents can be extracted without much thinking.
4. Inheriting from the super Message class: Once I had a standard Message class, I had difficulty holding content for all types of messages. Thus, I inherited from the main Message class and the inherited classes have the specialized information. E.g. the ExceptionMessage holds the exception that occurred and the ReturnMessage holds the returned object. We have 3 kinds of messages:
   1. InvocationMessage: invoking a remote method
   2. ReturnMessage: returning results from server
   3. ExceptionMessage: returning exceptions (I encapsulate them into Remote440Exception later)

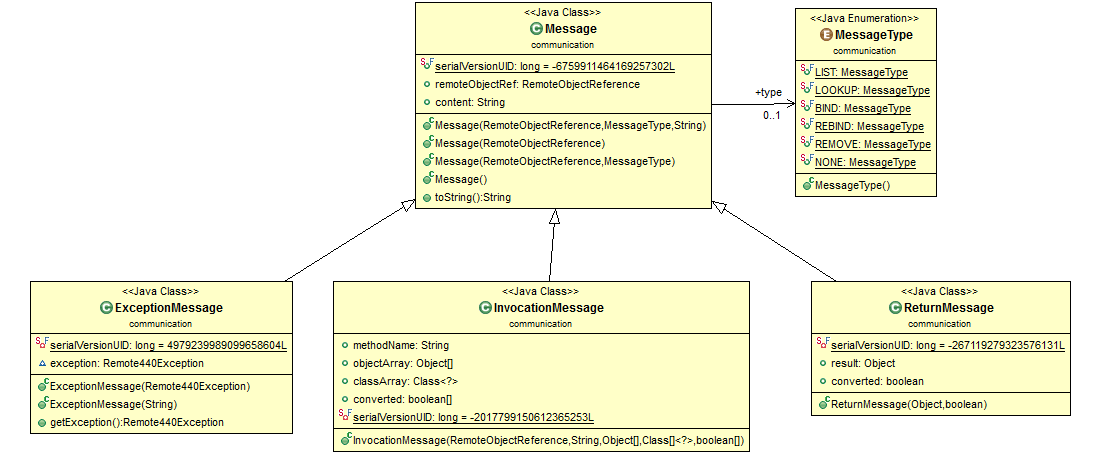


Figure 2: Message SuperClass and its Children

The figure above shows how different messages are derived from a single Message class.

I was also planning on a **MissingClassMessage** to hold the actual class file that needs to be transferred over the network when we require a missing “.class” file in RMI. This is a do-able task but I am limited by time.

1. Passing RemoteObjectReferences as parameters: When I was doing a final overview of my project, I realized that my framework was passing RoRs implicitly. But later, on looking at the write-up, and referring the ZIPCodeServer example, I realized that RoRs also need to be sent explicitly. The problem with that would be that the explicitly passed parameters would be converted to local objects at the server. This was a caveat that I had missed. I corrected it by storing Boolean values that signify whether the RoR needs to be converted into a Local Object at the server. The same for return message for the reverse process.

HOW TO RUN MY PROGRAM:

Go to “/cmu.ds.lab2/bin/“

1. Start registry server

java registry.RegistryServer

1. Start server

java server.Server <registryIP> <registryPort>

If not given registryIP and registryPort, it starts with default values

1. Start client

Java client.Client <registryIP> <registryPort>

If not given registryIP and registryPort, it starts with default values

1. Start Zip..

Create object:

Enter Class Name (example1.Calci OR test1.ZipCodeServerImpl OR test2.ZipCodeRListImpl OR test3.NameServerImpl):

test1.ZipCodeServerImp

Now object is created. We use this object for RMI..

Java test1/ ZipCodeClient <registry\_IP> <registry\_port> <object\_name> <outputfile>

java test1/ZipCodeClient 127.0.0.1 1099 Zip1 ../src/test1/data.txt

The source files are inside “/cmu.ds.lab2/src/”

I have provided sample sample test cases (option 3 on the both, server and client) to create sample objects and register them into the registry so that they can be looked up conveniently.

Stub Compiler

The stub compiler should be run on the interface of the concerned class file.

IMPORTANT: Suppose the name of the class file is Calci.java, then the name of the interface should be CalciInterface.java and stub will eventually be named CalciInterface\_Stub.class

EXCEPTIONS

ROR NOT EXPOSED

Examples: