Report of RMI Facility
Project 2 of 15-640
Kailiang Chen(kailianc) & Yang Pan(yangpan)

Design

In this project, our goal is to design and implement a **RMI(Remote Method Invocation)** facility for Java, which provide a mechanism by which objects within one Java Virtual Machine (JVM) can invoke methods on objects within another JVM, even if the target object resides within a JVM hosted by a different, but network accessible, machine.

As we know, **remote procedure call (RPC)** approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.

Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations. As in Figure 1 and Figure 2, RMI is a middleware layer framework, which is based on Request-Reply protocol.

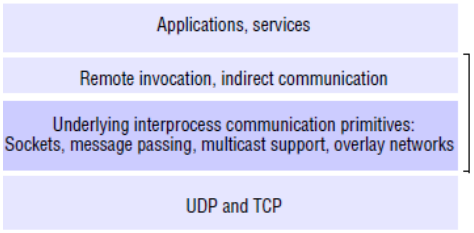


Figure 1 Middleware Layer

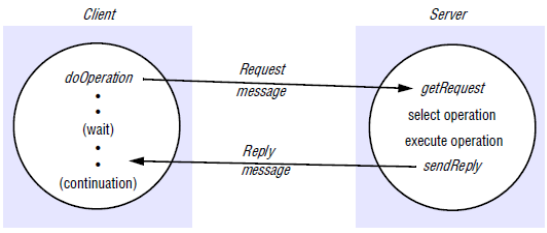
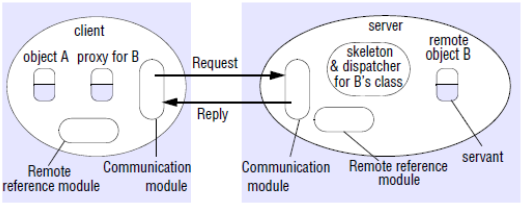


Figure 2 Request-Reply Protocol

Standard Java RMI framework, which realizes a **Distributed Object Model** using Java Socket, Reflection and Serialization mechanisms. It is made up of a communication module, a remote reference module and some other parts like proxy, dispatcher and skeleton, as Figure 3 and 4 shown.



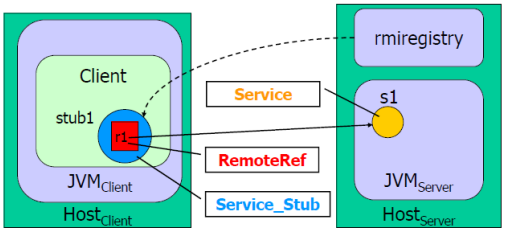


Figure 3 Role of proxy and skeleton

Figure 4 Stub and Remote Reference

The key to **RMI** is **proxy** pattern, which is based on an idea that interface defines behavior and class defines implementations (Figure 5). It is built from three abstraction layers, Stub and Skeleton layer, Remote Reference layer and Transport layer (Figure 6).

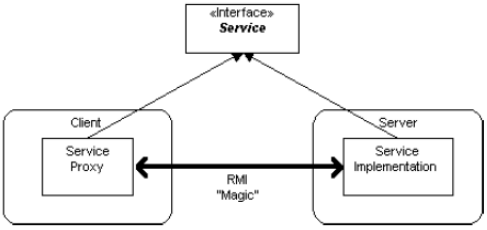


Figure 5 Proxy Pattern

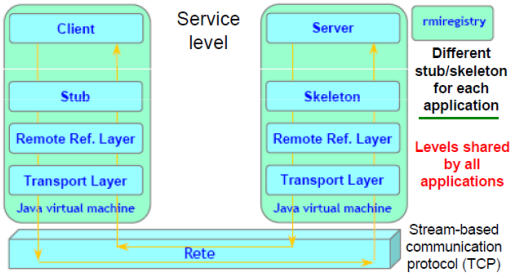


Figure 6 RMI Architecture

The whole RMI workflow is:
(1) A server object is registered with the RMI registry; it runs on server that hosts remote service and accepts queries for services, by default on port 1099.
(2) A client looks through the RMI registry for the remote object by lookup();
(3) Once the remote object is located, its stub is returned in the client;
(4) The remote object can be used in the same way as a local object. The communication between the client and the server is handled through the stub and skeleton.

Our RMI facility mimics the fundamental functions of standard Java RMI. It has a communication module, a remote reference module, a stub interface for different kinds of service to implement and a dispatcher. More specifically, our RMI has Remote Registry, Remote Object Reference, Remote Interface, Remote Exception and Actions in distributed system environment. However, we have no consideration on distributed garbage collection, the .class files download system and rmic compiler system.

Moreover, the basic design decision might be on the Security and Scalability of RMI system, which means server has right to authorize Client to invoke some actions and multiple Clients can manipulate the same remote object at the same time. In standard java, we can find a SecurityManager class to provide security protection mechanism. Here, we have not implemented it as we figure the purpose of this project is to mimic basic remote invocation function in RMI. And if we want to consider Scalability problem, synchronization control might be needed.

UML

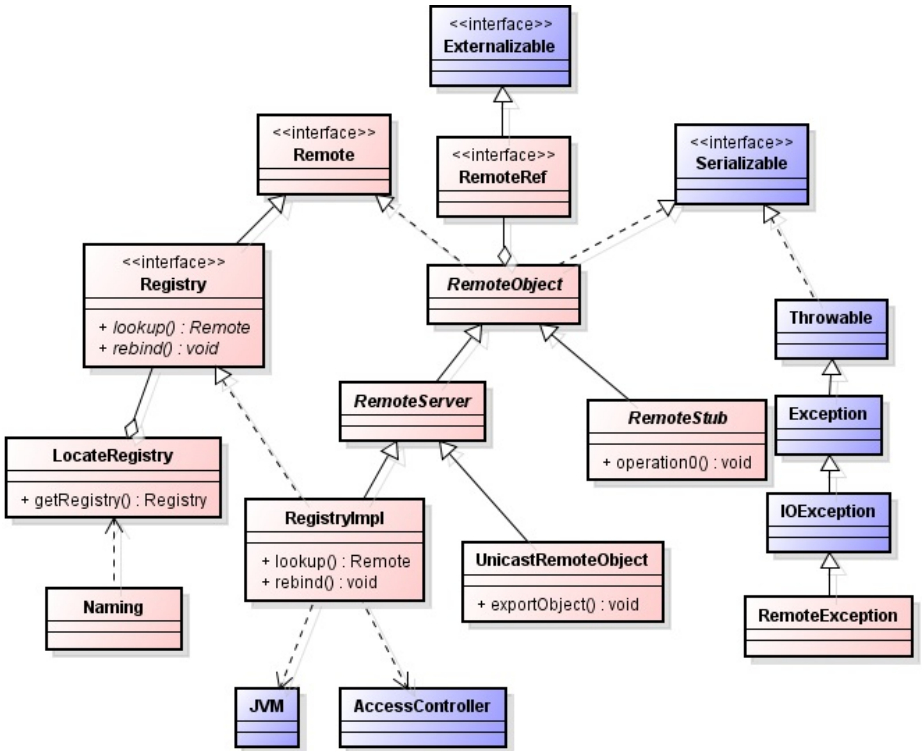


Figure 7 Java RMI Diagram

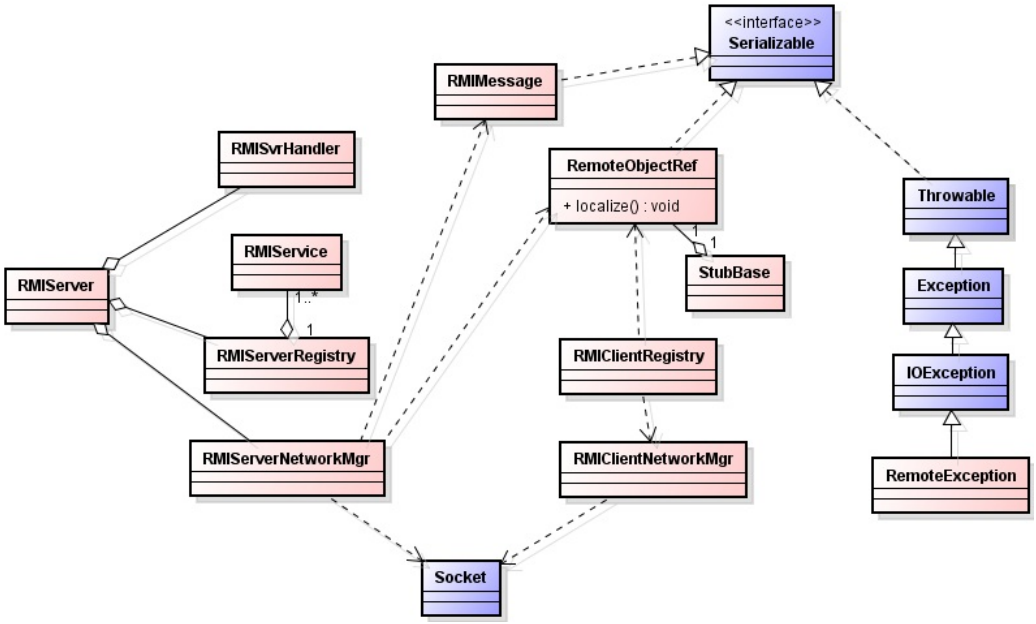


Figure 8 Our RMI Class Diagram

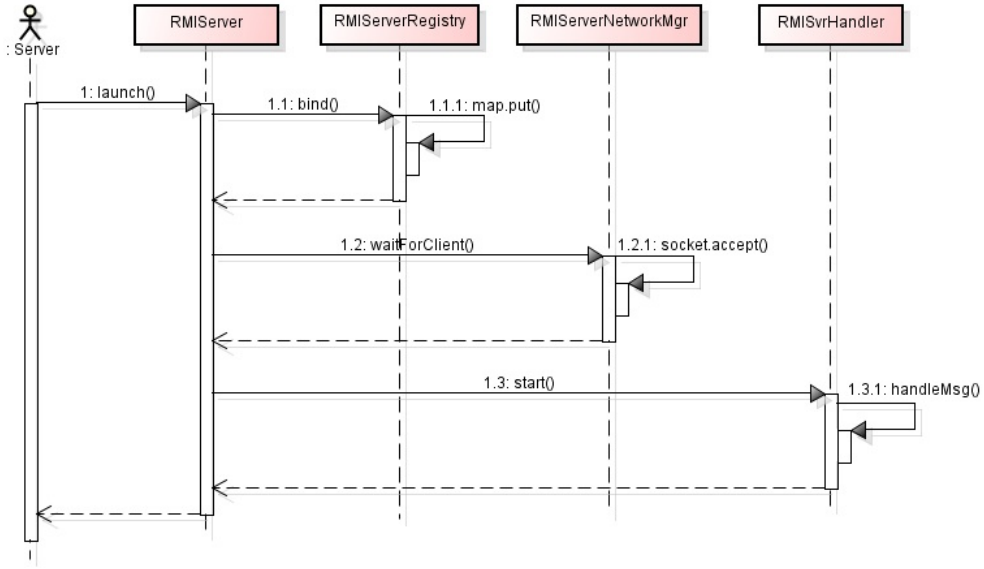


Figure 9 Server Sequence Diagram

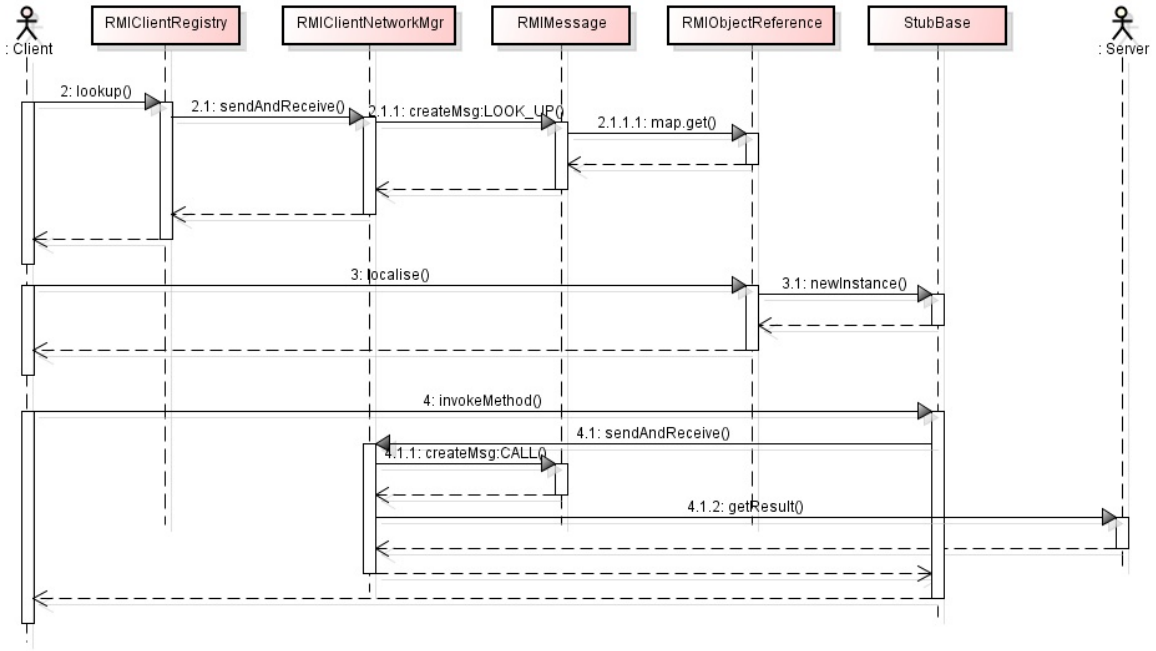


Figure 10 Client Sequence Diagram

Special features

1. Support RMI(Remote Method Invocation) between Server-Multiple Clients
2. Support object, primitive type as parameter and return value
3. It's an easy-to-use facility framework as simple as standard java RMI, you can export "rmi" package to everywhere to develop the RMI project.
4. It's an easy-to-extend and loosely coupled communication module and remote reference module for Server-Client

Implementation

Class Implementation

According to the UML class diagram, our system consists of:

1) **RMIMessage**: a concrete class defines the message transmitted between Server and Client, which is used during service lookup, method invocation and result return. Additional message type and content can be added if necessary.
2) **RemoteObjectReference**: a concrete class defines the service instance reference Server provides for Client, and server manages its information.
3) **RMIService**: an abstract class defines interface for services and serves as an element of Server's information table. It supports pass by reference and pass by value depending on whether Client service extends it or not.
4) **StubBase**: an abstract class defines interface for services and wraps RemoteObjectReference for Server/Client communication.
5) **RemoteException**: a common exception class defines the specific exception for RMI facility, it can be extended by adding some message display or handling method in the future.
6) **RMIServerRegistry/RMIClientRegistry**: registry classes define the basic service lookup and bind functions for Server and Client. A common base class or interface can be added to improve the extensibility, however, we leave it unimplemented, as it is not necessary in our project.
7) **RMIServerNetworkMgr/ RMIClientNetworkMgr**: communication classes define the basic message transmission functions between Server and Client. Similarly, we divide it into Server/Client module as we do for registry classes for readability.
8) **RMISvrHandler**: Server monitor thread for accepting messages from a client. It accepts message and passes it to RMIServerNetworkMgr to process.
9) **RMIServer**: Server main thread to bind a new service and launch monitor thread.
10) **RMIClient**: Client main thread to lookup, localize and invoke a new service.
11) **ServerConst/ClientConst**: constant parameters defined
12) **Student/StudentList**: concrete service classes of our test cases defined on Server-side
13) **Student_stub/StudentList_stub**: stub classes of our test cases used on Client-side

Development environment

This project is developed with Eclipse IDE for Java Developers, Luna Release (4.4.0), JDK 8u20. If you want to write a test class inheriting RMIService, you should work in the same environment.

Test with our examples

We provide two service classes to test the RMI facility and other features. They are Student and StudentList.

Student is a simple service class with name and score set/get functions.
StudentList is another service class with student register and first student display functions. The purpose is to test the case when another service instance is used as return value.

Test environment

A server at unix.andrew.cmu.edu, unix machines connected to AFS

Deploy and run

1. Open two Terminals and use one as Server, the other as Client.
2. Check the Server IP address, and set the Server IP address in the ./src/client/rmi/client/ClientConst.java. And if needed, set the port number in the following setting files:
   ./src/client/rmi/client/ClientConst.java
   ./src/server/rmi/server/ServerConst.java.
   (Default IP address is 127.0.0.1 and port number is 1099)
3. Navigate to ./src/server on one Terminal and the other to ./src/client. (For convenience, we name the first one as “Server Terminal”, the second one as “Client Terminal”).
4. Type “make” on Server Terminal and Client Terminal.
5. On the Server Terminal, type “make run” to launch RMIServer
6. On the Client Terminal, type “make run” to launch RMIClient. (Warning: you should run server program first, and then client program)
7. The result will display on the Client-Side.

Understand the test result

In the client terminal, we can see four lines of output:
Before name modify: Yan Pan has score 100
After name modify: Yang Pan has score 100
After score modify: Yang Pan has score 80
Finally: Yang Pan has score 80

You can check out our comments in the source code of “RMIClient.java” to see what’s going on here. In this report we will just have a brief explanation.

The first line is a look inside a remote object reference of Student class. It’s the return value of another remote invocation (StudentList.getFirstStudet()). Because Student is a subclass of “RMIService”, it is passed by reference.

“Yan Pan” and “100” are the return values of two remote invocations. They are passed by value because they are not subclass of “RMIService”. Here we can see the original value.

The second line is the result of another remote invocation to change the name of that student. In the same fashion, the third line is the result of a remote invocation to change the score of that student.

To confirm again that the student is passed by reference, we remotely invoke StudentList.getFirstStudet() again to get the student and check its name and score. That’s the output of the fourth line. It confirms that the class is passed by reference.

Writing customized test class

By using the RMIService class on Server side, you can write your own service class to provide functions, and then by using StubBase class, you can create a service stub class on Client side (In real scenario, it should be created on Server side automatically and transmitted to Client to use). Then, you can use service functions on Server side by adding service lookup and method invocation code in Client main function.

Developer environment

You can write the class anywhere you want, as long as the service class in the server inherits from RMIService and stub class in the client inherits from StubBase. However, it's strongly recommended to develop in the same environment as ours.

Understand our interfaces and framework

Our frame is almost the same as the standard Java RMI. But the stub is slightly different. You can refer to the "StudentList_stub.java" and "Student_stub.java" to develop your own stub files. When finished, put it to "./RMI_640_Client/src/stub".

The service class in the server is easy to develop. You just need to extend it from "RMIService" to your finished class and put it to "./RMI_640/src/services".

Deploy and run

Once you finish your work, copy your service class file (say, ZipCode.java) to the directory "./src/server/services/". And then, edit ./src/Makefile by adding "./ src/server/services/ ZipCode.java" to "CLASSES".

Now, change directory to ./src/server/ and type the following commands to run the whole program:

> make
> make run

On the client, you should copy stub class to the directory "./src/client/stub/". And then, edit ./src/Makefile by adding "./src/client/stub/ZipCode_stub.java" to "CLASSES". Now, change directory to ./src/client/ and type the following commands to connect to server and run the client program:

> make
> make run

Have fun with our nicely framework~