15640 Project2 Report

Design and Implementation of an RMI Facility for Java

Name1: Jian Wang
AndrewID:jianw3
Name2: Yifan Li
AndrewID: yifanl

Part I: Features

The RMI facility we have implemented has following features:

1. The remote object reference interface

We have created a remote object reference interface for application developer to name their own remote object.

2. The remote method invocation

The ability to invoke the remote object's method is enabled in our RMI facility. You can also invoke a method which would return the remote object. The ability to pass a remote object reference as an argument is possible in our facility but we got no time to implement it.

3. The ability to locate remote objects, e.g. a registry service

We have created a locate registry service to find the registry host. The registry itself is created as an independent part of the facility which makes it possible to serve more than one remote reference look up.

4. Using dispatcher instead of skeleton

We have implemented the server side using dispatcher model to listen to the method invocation requests from the clients. In this way, we don't have to create a skeleton for each remote object reference. A table of local objects and remote object reference keys are maintained on the server.

Part II: Design

The whole structure of our design could be showed in the figure 1.

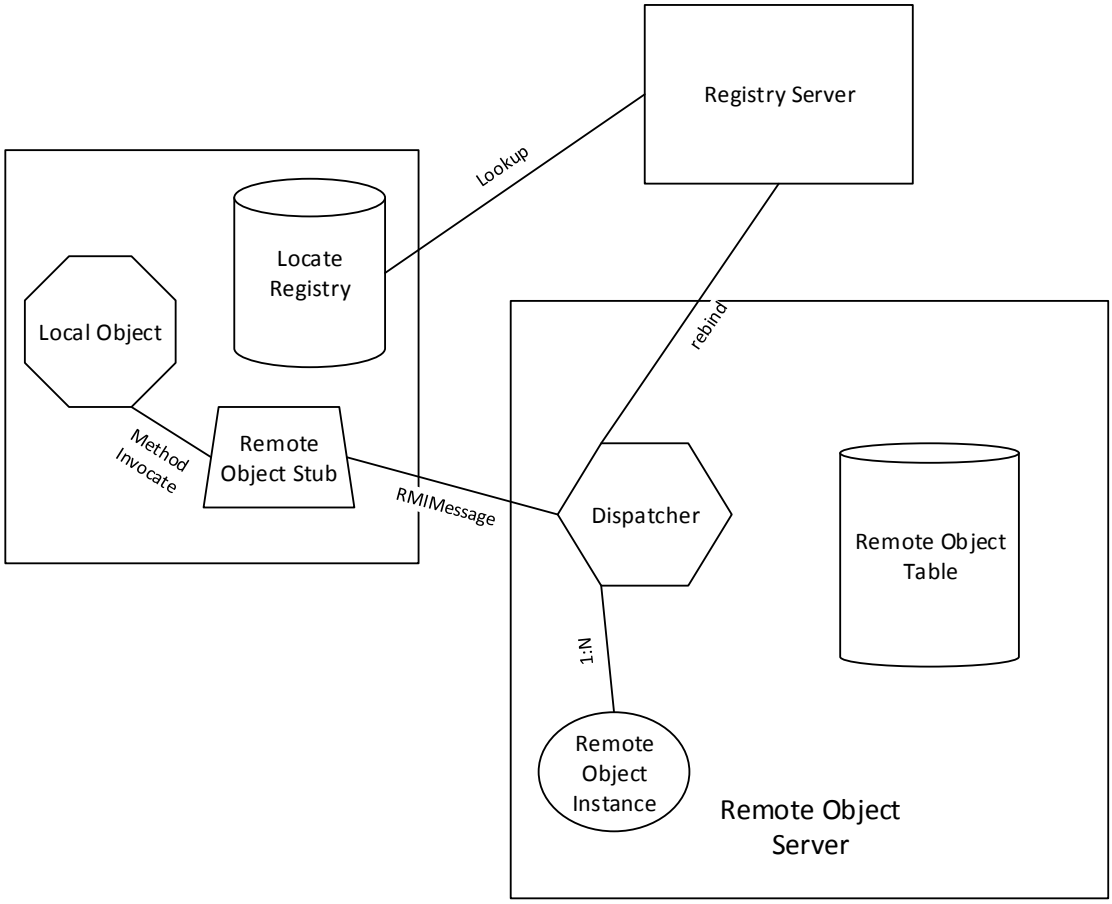

Figure 1 RMI System Architecture

The whole RMI facility is divided into three parts: Client, Remote Object Server, and Registry Server.

The Registry Server is responsible to maintain a service table accessible in the remote object server. Client could loop up the service and get the remote object reference from the registry server. Remote Object Server could bind and rebind the name of the method to the service name on the registry server. It is possible for the registry server process multiple requests at the same time.

The Remote Object Server is the place the real method invocation happens. The related object on the server would invocate the method and return the value to the client. If the return value is a remote object. The server would return the remote object reference back instead of the remote object class to the client. Dispatcher is the component responsible to listen to the server socket and invoke new threads to process the request.

The client should firstly locate the registry server and get the remote object reference first. Then the remote method invocation would execute through the stub on the client. The stub here is coded by hand instead of producing automatically.

Application developer could use our RMI facility by simply just implementing our provided RemoteInterface Java interface. It's highly recommended to check our test cases in the fourth part to learn how to use our facility.

Due to the lack of time, we haven't implemented following parts:

1. A stub compiler. We hard-coded the stub for the client. However, it is also possible to write a stub compiler to produce the stub automatically.
2. The automatic retrieval of .class files for stubs.
3. A distributed garbage collector.

Part III: Deployment

Require: JRE Java 1.7, Linux, AFS file system

1. Copy source code to each machine
2. "cd" to the src/ directory
3. "make"
4. "cd" to the **bin** directory
5. For server machine:
   Start Registry
   Type in: java server.RealRegistry <port number>
   Start RMI server
   Type in: java server.RMI_Server <IP address for the registry> <port number for the registr> <remote class name>
   Example: java server.RMI_Server 128.237.216.75 8083 example.ZipCodeServerImpl
   Or
   java server.RMI_Server 128.237.216.75 8083 example.ZipCodeRListImpl

6. For client machine:
   Type in: java example.ZipCodeRListClient <IP address for registry> <port number for registry> <remote class name> <input file name>
   Example: java example.ZipCodeRListClient 128.237.216.75 8081 example.ZipCodeRListImpl ../zipCode.txt
   Or
   java example.ZipCodeClient 128.237.216.75 8081 example.ZipCodeServerImpl ../zipCode.txt

Part IV: Test Case

1. Example:
   a) ZipCodeServer: This test case will call the remote ZipCodeServer object to find the zip code for a city.

      - **Start:** start registry and RMI sever first

Type in:

java server.RealRegistry <port number>

java server.RMI_Server <IP address for the registry> <port number for the registr> <remote class name>

Example:

java server.RealRegistry 8083

java server.RMI_Server 128.237.216.75 8083 example.ZipCodeServerImpl

- **Test Client:** use ZipCodeClient to call the ZipCodeServer remote object

Type in:

java example. ZipCodeClient <IP address for registry> <port number for registry> <remote class name> <input file name>

Example:

java example.ZipCodeClient 128.237.216.75 8083 example.ZipCodeServerImpl ../zipCode.txt

b) ZipCodeRList: This test case will call the remote ZipCodeRList object to add and look up zipcode. The add and next method will return a RemoteObjectRef to the client.

- **Start registry and RMI sever first**

Type in:

java server.RealRegistry <port number>

java server.RMI_Server <IP address for the registry> <port number for the registr> <remote class name>

Example:

java server.RealRegistry 8083

java server.RMI_Server 128.237.216.75 8083 example.ZipCodeRListImpl

- **Test Client:** use ZipCodeRListClient to call the ZipCodeRList remote object

Type in:

java example. ZipCodeRListClient <IP address for registry> <port

number for registry> <remote class name> <input file name>

Example:

java example.ZipCodeRListClient 128.237.216.75 8083 example.ZipCodeRListImpl ../zipCode.txt