15-640 Project2 Report

Name 1: Rui Zhang
AndrewID 1: ruiz1
Name 2: Jing Gao
AndrewID 2: jinggao

Part I: Design and Features

The program implements the RMI facility. The framework follows the suggested big picture.

There will be one RMI registry server, several remote servers, and several clients. The registry server takes care of RMI registry, and processes requests (“rebind”, “lookup”, “list”) from clients or remote servers. The remote servers send requests to registry server in order to register new remote services, and process RMI from clients and respond the return values to them. The clients look up remote services from the registry server, call methods from client programs; the seemingly local method invocations reach client-side stubs of corresponding remote objects, which then communicate with the destination remote servers and finish the RMI.

On each registry server, remote server or client, there is a communication module taking care of connections with other clients/servers, recording destination information like ip address (host name) and port number, caching corresponding sockets, and sending/receiving messages from the sockets when requested by its host. Communication medium is various types of RMI messages, including registry message, ROR message, method message, return value message, and exception message. They carry various types of information like method info and registry info, among the RMI systems.

Remote objects are referenced by remote object references (ROR); they include host information, unique object key and local references on remote servers. ROR contains localise method which will create an instance of local stub for remote object. Stub classes for remote objects can be generated from stub compiler which will compile the stub classes from original remote objects. The client program gets the ROR from looking up registry, localizes a stub with corresponding ROR and calls the method on stub to invoke the remote method. The method called on the stub

will communicate with destination remote server via method message and return value message and finish the RMI.

Remote exceptions and failures during the processes of RMI are captured locally on the remote servers, capsuled in the exception messages and sent back to the calling clients. The process is as if exceptions are thrown locally on the clients.

The program comprises of the following major classes.

A. Communication class group:

A1. CommunicationModule: This is a class responsible for communication with remote clients or servers. On each client/server, it keeps a HashMap of hostname/port info and corresponding socket. Givin remote host name and port, it can read from or write to the corresponding socket

A2. Remote: This is the interface for remote object. This could be RMI message sent from remote clients/servers, remote object reference, normal remote objects, etc

A3. RemoteObjectRef: This class is reference of remote object. It contains necessary information about remote object, like ip address, port number, object key and remote object class name. It can localise on a remote client from the original object, and create an instance of the stub there. Providing remote object reference, method on it, and parameters, the stub will pass the information via method message, invoke the remote method on a remote server, and return the return value from the remote method

A4. RMIMessage: This is the interface for all types of messages among RMI clients and servers, including ROR messages, exception messages, etc

A5. Messages: The following message classes implement RMI interface.

RorMessage – carrying remote object reference

RegMessage – carrying regInfo object which contains the command enum (REBIND, LOOKUP or LIST), service name and remote object reference that will be sent to RMI registry

MethodMessage – carrying MethodInfo object which include object key of the object which possess the method, method hash code and method parameters

RVMessage – carrying return value of RMI.

ExMessage – carrying remote exceptions

A6. RemoteStub: This class contains the corresponding remote object reference.

B. RMI Registry Server class group:

B1. Registry: This is the interface for RMI registry operations, which include lookup, rebind and list methods. It extends Remote and Serializable interfaces

B2. Registry_Server: This class works as the RMI registry server. It keeps a <serviceName, remoteObjectReference> hash table. It contains the main() method running on the RMI registry server, which listen on a port number and forward a new connection to a new registryThread that will parse the request and pass corresponding jobs like lookup, rebind and list on the registry hash table. The above mentioned jobs are realized by implementing Registry interface

B3. registryThread: This is a class responsible of serving a connection to the RMI registry server. It adds the connection to CommunicationModule, then keeps reading requests (commands) from the connection, passing the corresponding operations on RMI registry to RMI registry server

B4. Registry_Client is a class responsible for remote visits to the RMI registry server from its clients (either RMI clients or RMI servers). It implements Registry interface. From the clients, operations on remote RMI registry are like local operations

B5. LocateRegistry: This is a class responsible of locating RMI registry. It generates a Registry_Client object connected to the Registry_Server.

C. Remote Server class group:

C1. RemoteServerRef: This is the remote server where remote objects and methods reside on. It keeps a <objKey, Object> hash table, a <objKey, objMethods> hash table, and a <Object, objKey> hash table. The main() function runs on the remote server, which creates a remote thread to listen to connections and then scan for user command to add remote object and its methods to its hash tables and send corresponding rebind request to RMI registry server in order to register new services there. At the same time, do_job function deals with method messages passed from remote services, performing the core functionality of invoking remote methods relative to the clients, and returning the return values of the remote methods

C2. Remote_Thread: This is runnable class running in a thread of remote server, responsible for continuously accepting connections from clients. Upon a new connection is established, it creates a corresponding Remote_Service which will serve the connected client

C3. Remote_Service: This is a class responsible of serving a connection to the remote server. It adds the connection to CommunicationModule, then keeps reading RMI method messages from the connected client, passing the corresponding message along with client hostname and port number to the remote server do_job method, where the method message will be processed

D. Client class group:

D1. Client classes: The client programs that will invoke RMI by calling the methods in corresponding stub.

D2. Stubs: RMI stub compiler generated or manually generated stubs for remote objects

D3. rmic: This is the stub compiler which will generate the _Stub.class file.

Part II: Implemented vs Umimplemented/Bugs

A. Implemented:

A1. the ability to name remote objects

implemented with remote object references and Registry service

A2. the ability to invoke methods on remote objects, including those methods that pass and/or return remote objects references and those methods that pass and/or return references to local objects

implemented via remote objects stub, ROR, method/returnvalue messages, and server mechanisms

A3. the ability to locate remote objects, e.g. a registry service

implemented with RMI registry server, registry message, and the returning ROR message

A4. a stub compiler

implemented in rmic class

A5. the automatic retrieval of .class files for stubs

If the .class files for corresponding stubs doesn't exist locally, it will send a request message to the remote server. The server then retrieve the .class file and send it to the client. The client then store it as a .class file. The file content is transferred in binary stream.

B. Unimplemented:

B1. a distributed garbage collector

design: We could maintain a reference count in the server side for each remote object. When one client gets a remote reference of one specific remote object, the count increase by 1. And there should be a lease for a client toward this remote object, if the client wants to use it for a period time, it should periodically send renew request to renew its lease. Otherwise, the server will consider that the client is disconnected. When the lease expires, it will decrease the reference count by 1. When the reference count reaches 0, the remote object will be cleaned.

Part III: How to Build, Deploy and Run

1. copy program to each server/client
2. cd into RMI directory
3. ant build
4. cd into bin directory

5. java rmic.rmic a remote object class such as ZipCodeR.ZipCodeRListImpl or Examples.ZipCodeServerImpl or nameserver.NameServerImpl

6. java Registr.Registry_Server [registryPort] (do this on RMI registry server if you haven't input the registryPort the program will use the default port number 15640)

7. java Server.RemoteServerRef [serverPort] [registryHostname] [registryPort] (do this on remote server if you haven't input the corresponding any arguments, the program will use the default server port:15440, default registry hostname:localhost and default registry port:15640)

8. After lanching the remoteServerRef, it will show the command line.

Like this: -$->

And you can input like this: [Remote Class name] [service name]

The remote object class name adding the service name decided by user.

Here, we can use Examples provided by us ZipCodeR.ZipCodeRListImpl or Examples.ZipCodeServerImpl or nameserver.NameServerImpl .

9. java Example.aProgramClient [registryHostname] [registryPort] [aService] [filename] (do this on client)

Part IV: Dependencies and Software/System Requirements

Linux, Java 1.7.0, ant

(All of the requirements are satisfied on Andrew Linux machines)

Part V: Test Framework with Three Examples

See how to build and run our program as well as the inputting arguments, please see **Part III.**

Example 1. ZipCode

cd into RMI

ant build

on RMI registry server:
cd into RMI/bin
java Registry.Registry_Server 15640

on remote server:
cd into RMI/bin
java Server.RemoteServerRef 15440 localhost 15640
-$->Examples.ZipCodeServerImpl zcService

on client:
cd into RMI/bin
java rmic.rmic Examples.ZipCodeServerImpl
java Examples.ZipCodeClient localhost 15640 zcService data.txt

Example 2. ZipCodeR (mainly test returning a remote object which is pass-by-reference)

cd into RMI
ant build

on RMI registry server:
cd into RMI/bin
java Registry.Registry_Server 15640

on remote server:
cd into RMI/bin
java Server.RemoteServerRef 15440 localhost 15640
-$->ZipCodeR.ZipCodeRListImpl zcRService

on client:
cd into RMI/bin
java rmic.rmic ZipCodeR.ZipCodeRListImpl
java ZipCodeR.ZipCodeRListClient localhost 15640 zcRService data.txt

Example 3. NameServer(mainly test passing a remote object argument which is pass-by-reference)

cd into RMI
ant build

on RMI registry server:
cd into RMI/bin
java Registry.Registry_Server 15640

on remote server:
cd into RMI/bin
java Server.RemoteServerRef 15440 localhost 15640
-$->nameservere.NameServerImpl nserver

on client:
cd into RMI/bin
java rmic.rmic nameserver. NameServerImpl
java nameserver.NameClient localhost 15640 nserver