15-640 Lab 2 Remote Method Invoke

Yuning He yuningh@andrew.cmu.edu
Peixin Zheng peixinz@andrew.cmu.edu

Design and Use

This project implements a Remote Method Invoke framework and several example classes for testing purpose. We have implemented some handy test codes. We also accept users' commands as for testing.

This project uses a server-client structure. On the client side there is a proxy for communicating with the server at the beginning, and building up stubs for each remote object. After it is established, a stub is responsible for marshalling RMI requests, sending it to the server, receiving response, and unmarshalling the response to get the return value. On the server side, the server generally listens to the specified port for RMI requests. Once it receives a message from the client, it either returns a remote object reference or invokes the specified method and sends back the return value according to the request type sent by the client.

Figure 1 shows the framework of our design.

We implement a *RemoteObjectRef* class, which is used for describing and localizing a remote object reference on the client side. *RemoteStub* class is the base class for all stubs, inside we implement the *execute()* method which includes the communication part of remote method invoke. Each remote object has its own stub class (e.g. *RMIExample* class has *RMIExample_stub* class as its own stub), and each stub holds the *RemoteObjectRef* of the object it's representing. For communication, we have the *Message* class for both the initialization part and the method invoke part.

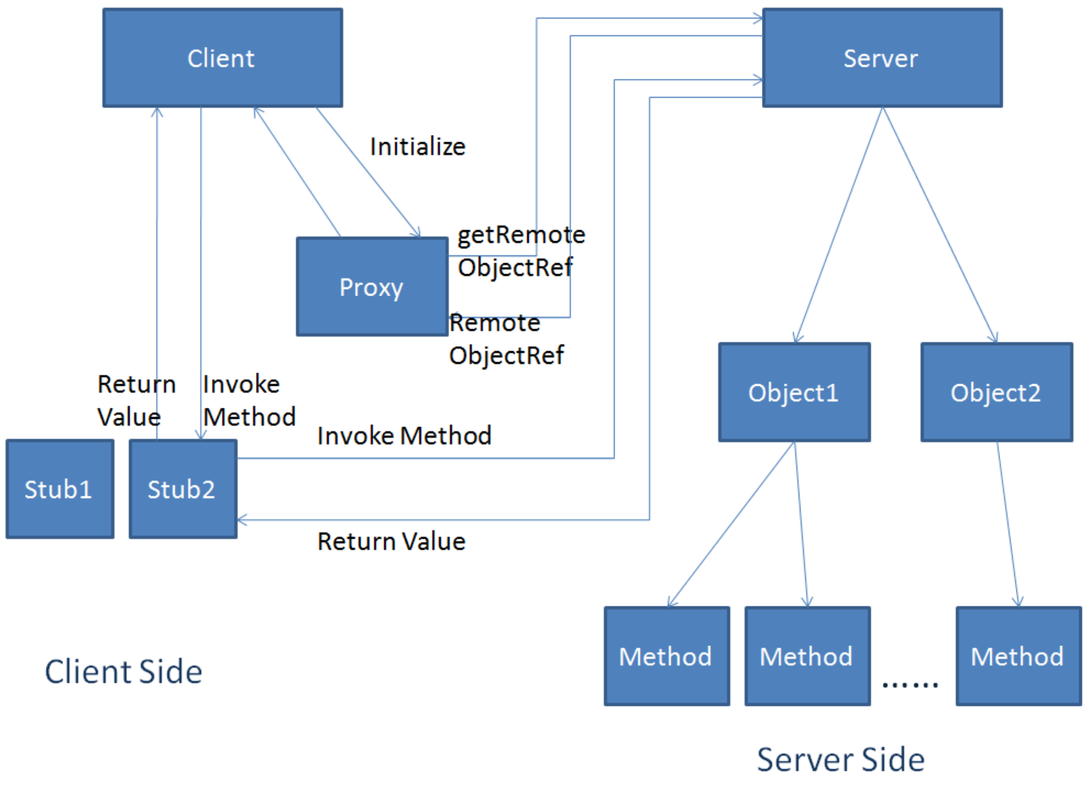

Figure 1. RMI framework

Upon starting up the system, the server sets up all the remote objects that are going to be used. Then the server listens to the specified port, waiting for requests from the client. The client will first ask for all the *RemoteObjectRef* of the remote objects, and then it will localize these references, creating a stub for each of them. The name of the stub is strictly related to the name of the remote object, so that we know where to look for the stub according to the name.

When the client decides to invoke a remote method, it uses the stub to call this method. A stub has all the methods of its corresponding object. For example, there is an *add2()* method in *RMIExample*, then *RMIExample_stub* also has an *add2()* method. However, these two methods do not do the same thing. The method in the stub actually prepares the *objectName*, *methodName*, and arguments, then send these information to the server. It is at the server side that the method is really invoked. Server takes the input arguments and applies them to the local method. Then after execution, the return value is sent back to the client, so that it seems exactly the same as local method invoke. When the client passes a reference as an argument, the corresponding stub actually passes the *RemoteObjectRef* as an argument. On the server side the server will first check the argument type. If it is a *RemoteObjectRef*, server will then look up for the local object and invoke the method with it.

We have a *Message* class so that communication can be well organized. We have included both *RemoteObjectRef* and method-related stuff inside *Message*, so that one single *Message* class can be used in both initialization stage and method invoke stage.

For testing, we have enabled both built-in testing and testing according to user commands. After launching the system, first we do all the built-in testing. Then you can test these methods yourself. All the testing methods are in *RMIExample* and the corresponding stub. *RMISubExample* is provided for you as a tool to test the "pass by reference" part. Its stub is just used as an input argument in *RMIExample_stub* methods. The methods in *RMISubExample* have the same functionality compared to *RMIExample*. So you can just test the methods in *RMIExample*. *RMIPerson* is just a toy class used to test "return by reference" part. It only contains a string field and an int field for name and age property.

We provide the following methods for testing in *RMIExample*.

Method name	**Arguments**	**Return value**	**Functionality**
add2	int, int	int	add two numbers
add3	int, int, int	int	add three numbers
concat2	string, string	string	concatenate two strings
concat3	string, string, string	string	concatenate three strings
length	string	int	calculate length of string
incAttribute	null	void	increase the attribute in RMIExample by 1
getAttribute	null	int	get the attribute in RMIExample
incAttributeByRef	stubSub, int	void	increase the attribute in a reference by a given

			number
getAttributeByRef	stubSub	int	get the attribute in a reference
getPerson	null	RMIPerson	get an RMIPerson object
modify	string, int	void	modify the attributes in an RMIPerson object

Table 1. Testing methods

Portion of Implementation

This project has implemented all the required functions of Project 2. To be specific,

1. Our system can get remote object references in the *Proxy* class. The proxy sends message to the server specifying the object name, and then it can get the *RemoteObjectRef* back.
2. Our system supports remote method invoke by implementing stubs. When the user wants to invoke a method in an object, we use the stub corresponding to the object to send message to the server. After server has done the actual invoke work, return value is sent back.
3. Our system has a built-in registry inside the server. The server holds a map of object names and real objects. The client can get remote object references from here.
4. We have implemented other functions such as call by reference. We can interact with user and invoke remote methods according to the user's instruction.

However, there are some extensive functions that we do not implement:

1. We do not have a stub compiler. Our stubs are hand-written.
2. We do not support the automatic retrieval of .class file.
3. We do not have a distributed garbage collection mechanism.

Build and Run

In the src directory there is a *RMIClient* directory and a *RMIServer* directory. Put the *RMIClient* at the client side and put the *RMIServer* at the server side.

At the server side:

```
~$ cd RMIServer/src
~$ make
~$ java RMIServer [listenPort]
```

Then the server will start running and wait for the client. You can see the IP address of the server in the output. You will need it when you start the client.

At the client side:

```
~$ cd RMIClient/src
~$ make
~$ java RMIClient [serverIP] [serverPort]
```

Then the client will start running. It will start doing the built-in test cases as soon as it starts running. After that, you will see a prompt like this:

```
Client>
```

You will also see some instructions on how to do your own testing. Feel free to test any method you want.

Testing Methods

When you have successfully built and run our program, you can know test our RMI framework. The program first runs some demos and shows the correctness of the RMI mechanism. Then it will lead you to a console where you can type in your own test cases.

As we said above, you can test all the methods in *RMIExample*. We don't provide the interface for you to test the methods in *RMISubExample* because it acts like a tool for you to test "pass by reference" part. We have created the necessary stubs in the program so you can just type in the method name with arguments and see the output.

For you own input, the format is “methodName argument1, argument2, …, argumentN”. The method name and corresponding arguments are showed above. For example, you can type in

```
Client>add2 2 3
```

Then you will see the output is 5.

The only two methods for testing “pass by reference” are *incAttributeByRef* and *getAttributeByRef*. The argument is the stub for *RMISubExample* named “stubSub”, which we have created in the program. So specifically you can type in

```
Client>getAttributeByRef stubSub
Client>incAttributeByRef stubSub int
Client>getAttributeByRef stubSub
```

Then you will see the value changes in the *RMISubExample* object. Noted that if you type in

```
Client>getAttribute
Client>incAttribute
Client>getAttribute
```

You can only see the value changes in *RMIExample* object because it is a “pass by value” fashion. It only changes the attribute value in the *RMIExample* stub.

The testing method that returns an object is *getPerson*. For example:

```
Client>getPerson
Client>modify string int
Client>getPerson
```

Then you will see the changes in the *RMIPerson* object.

The rest of the methods are all in a “pass by value” fashion.