15-640 Distributed System Project2 Report

Name: San-Chuan Hung, Zhengxiong Zhang
AndrewID: sanchuah, zhengxiz

How to compile the code?

type “make compile”

How to set up servers and clients?

Launch Simple Registry Server

type “make registry”

Launch RMI Server

type “make rmi_server”

Run client

type “make client”

How to test the code?

You need to open three terminal screens to test the code. In the first screen, type ”make registry”, which will launch Registry Server on port 8888. In the second screen, type “make rmi_server”, which will launch a RMI Server on port 10000. Last, type “make client,” which will invoke ZipCodeServer.initialise(), ZipCodeServer.find(), ZipCodeServer.findAll(), and ZipCodeServer.printAll() and show the results.

System Overview

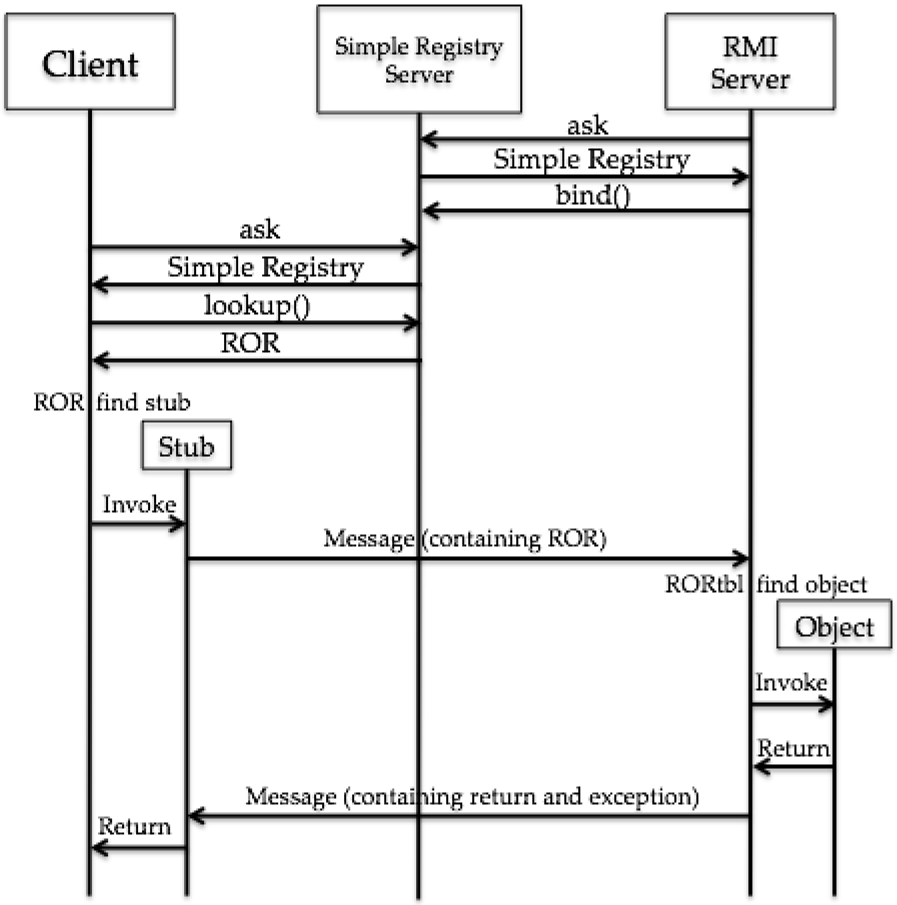

Figure 1. System Process Overview

The system contains mainly three parts: (1) Client (2) Simple Registry Server (3) RMI Server. Client is mainly responsible to interact with user, and to request remote method invocation to server side. Simple Registry Server is a bridge matching the requests from the client side and the corresponding RMI server. RMI server stores the worker objects in a table, invokes the methods in the object when the server receives requests from the client, and returns the results to the client.

The whole process of the system contains three parts: (1) registering Remote Object Reference (ROR) (2) requesting service (3) invoking remote method invocation.

First, Launch the Simple Registry Server, and then start the RMI Server, which will create a specified worker object in it’s ROR table, and registered the ROR object and the corresponding service name in Simple Registry Server; therefore, the client can find the ROR for the service through Simple Registry Server.

Second, the client requests ROR from the Simple Registry Server by the service name. ROR contains the address and the port of the RMI Server, and a method to localize a stub object. With the RMI Server's address information, the stub knows from where requests method invocation. After localises a stub object, the client can treat the stub object as a local object to invoke remote method and retrieve results.

Last, the client manipulates the stub to invoke remote methods and get results. After the client executes methods of stub, which will connect to RMI server, send requests of method invocation, get back the execution results from RMI server, and return the results to the client.

Design Decisions

In the Remote Object Reference hash table, we used ROR as the key and a new object as the value. So, we do not need to use skeleton any more. We do not need to generate skeleton any more, but just use Dispatcher to communicate with the stub. The advantage is that we map the ROR direct to the object with no more redundant processes. The disadvantage is that our system cannot handle big problems. If there are a lot of objects, our system maybe cannot process them well. If we have skeleton, I think we can use different ports for different objects.

Our Simple Registry have three methods: lookup(), rebind() and unbind(). We did not use bind() because out rebind() method can be considered as bind() + rebind().

We used a message class to transfer all the information between two machines. From the client to the server, it contains the ROR. From the server to the client, it contains the return object and the return exception.

If there is an exception happened in the server side, the server will send the object and the exception to the client. Then, the client will throw a RemoteException.

How to set up servers and clients using our RMI facility

1. Find three machines: client, RMI server and simple registry server.
2. Change the **RegistryHost** in the MakeFile. Copy the project in all machines.
3. make registry
4. make rmi_server
5. make client

General RMI Server

We design RMI Server which is general to all worker class. Although we provide ZipCodeServer as an example of service, but the RMI Server is not stick to ZipCodeServer class;

instead, RMI Server can support other classes, if they have implemented corresponding stub class.

The tradeoff is that it make the code inside RMI Server more complex, and it can be hard to add special mechanism for specific classes. For example, if some class needs special security checking before invocation, which can not be added into our RMI server. Our RMI server do not support customization.

Service Server and Stub

We use stub class and RMI server to implement remote method invocation. Stub class is a proxy of remote worker class in the client side. When the client side program invokes the method of stub class, which will send a message to the RMI server to request corresponding operation and wait for the return value from the server. When the RMI server receives the invocation request, it will find the corresponding object in it’s table, invoke the specified method of the object, and send back the result to the stub. Unlike stub-skeleton architecture, the responsibility of invocation in server side are mainly by RMI server.

The advantage of the design is to simplify the architecture, which does not need skeleton classes in server side.

The tradeoff of this design is that the logic of RMI server itself can be complex. RMI server has two roles: (1) connecting to Simple Registry Server for registering itself and the corresponding ROR (2) dealing with the request from stubs. If we extend more complex feature of invocation process in current architecture, like security checking, which will complicate the code of RMI server, and will increase the load of the RMI server.

Message Class

To reduce the complexity of communication between stub classes and RMI server, we design a message class, which contains ROR, the input variables (invocation methods and arguments), the return variable, and the direction of message(from stub to RMI or from RMI to stub). The message class can package all the variables needed by the both client and server, which can reduce the complexity of communication.

Conclusion

We propose a simple and general RMI implementation which mainly contains three parts: Simple Registry Server, RMI Server, and Client. To keep the whole architecture simple, we develop a message class for communication between the stub and RMI server, and merge the job of skeleton into the RMI server.

The developers can use our framework by implementing worker class and the corresponding stub class, and then our framework will take care of rest of work.