15-640: Distributed Systems Spring 2013
Project 2 Report

Yang Sun (Andrew ID: yksun)
yksun@andrew.cmu.edu
Yuan Gu (Andrew ID: yuangu)
yuangu@andrew.cmu.edu

Submitted: March 4, 2013

Overview

In this project, we've designed and implemented a basic RMI framework for Java, along with some test cases to verify the functionality of this framework.

This document is organized as following: first, we introduce the overall design and workflow of the framework. Then we review some implementation details and tradeoffs we consider valuable. We will continue talk about the tests applied to the implementation. Finally, we will how could our framework be further improved.

Design and Workflow

For the general design, we refer to "The Big Picture" from the handout. Our design consists of stub for remote objects on clients, registry on server and proxy dispatcher on server.
The framework consists of two sides, i.e. the server side and the client side. The basic workflow is:

Server	Client
Launch RMIRegistry Launch RMIProxyDispatch	
	Use RMINaming to look up for remote objects Invode remote methods through RMIProxyDispatch

Table 1: Workflow of the framework

RemoteObjectRef

Although RemoteObjectRef does not appear in the above workflow, it is an essential part of the framework and we will first of all introduce it.

RemoteObjectRef is the representation of a remote object through the whole framework. It contains the IP address, port where the remote object lies, as well as the class name of the remote object and the unique identifier (IDENTIFIER) of the remote object on the server.

Note that IDENTIFIER is a string specified by user. Thus, to access a remote object, both the client side and the server side should agree on the IDENTIFIER.

RMIRegistry

The RMIRegistry class runs on the server and serves as the registry for all remote objects on the server.

Once launched, the RMIRegistry listens on a given port and waits for two kinds of requests:

- *RMI_REGISTRY* is the registration request for a remote object. The request should contains a RemoteObjectRef instance. Once received this kind of request, RMIRegistry will save the RemoteObjectRef instance to a hashmap using IDENTIFIER as the key and the RemoteObjectRef as the value.
- *RMI_NAMING* is the request from the client to ask for information about a remote object. The request contains the IDENTIFIER to the related remote object, and the RMIRegistry will look up in the hashmap for the related RemoteObjectRef and return it back to the client.

RMIProxyDispatch

The RMIProxyDispatch class also runs on the server and serves as the proxy for any access to its related remote objects.

While launching, the RMIProxyDispatch creates some objects as is specified through the arguments. Then it registers these objects to the RMIRegistry. After this, it starts listening on a given port and waiting for one kind of request:

- *RMI_PROXY* is the request for invoking a method of a remote object. This kind of request is sent from the client to the RMIProxyDispatch running on the server. The request contains the IDENTIFIER to the remote object as well as the method name and the arguments for this method. The RMIProxyDispatch will try to invode the method on the related remote object and return the result and exceptions (if any) to the client.

RMINaming

The RMINaming class runs on the client. It provides the client a method called *lookup()* to use IDENTIFIER to look up a remote object on the server. If the remote object is found, *lookup()* returns a RemoteObjectRef instance to the client, or else, it returns null.

The RemoteObjectRef class provides the client a method called *localise()*. Once the client gets the RemoteObjectRef instance from *lookup()*, it needs to call localise() to get the stub for the remote object.

Implementation Details and Tradeoffs

While we do make lots of decisions or tradeoffs during the implementation phase, we would like to especially discuss two tradeoffs:

- Utilization of RMIMessage
- Seperation of RMIRegistry and RMIProxyDispatch

Utilization of RMIMessage

RMIMessage is a generic message class used for all kinds of communication in our framework, including the communication between RMIRegistry and RMIProxyDispatch, between client and RMIRegistry, and between client and and RMIProxyDispatch. We've found the use of a generic message class extremely valuable as it provides us a unified way to marshal and unmarshal the message, which greatly elinimates the amount of work we need to do on communication. It also enhances the robustness of the framework as we need not test the different kinds of communication seperately and could instead spend the time doing a lot of testing on the single message class.

The tradeoff we made in implementing RMIMessage is we directly use the ObjectInputStream and ObjectOutputStream to serialize and deserialize the RMIMessage class over the network. While using the ObjectInputStream and ObjectOutputStream might be slow, we do enjoy the simplicity and convenience it brings.

Seperation of RMIRegistry and RMIProxyDispatch

Another tradeoff we made is to seperate RMIProxyDispatch from RMIRegistry. In fact, all the work of RMIProxyDispatch could be done in RMIRegistry as RMIRegistry contains all the information about the remote objects. However, we choose to seperate the two classes as it is easier to implement and debug.

Test Cases

To figure out whether our design and implementation could fulfill the initial goals, we've provided some testing cases for this project.

HelloWorldNoArgs

The *HelloWorldNoArgs* is a very basic test case. It only provides a method called *sayHi()* and will return a string "Hi!" once called.

The main purpose for this test case is to test whether our implementation could support remote method invocation.

HelloWorld

The *HelloWorld* is an extension of the *HelloWorldNoArgs*. It also provides a method called *sayHi()*, but requires a string as the argument and will catenate the argument to the return value.

The main purpose for this test case is to test whether our implementation could support passing-by-value arguments.

HelloTimeout

The *HelloTimeout* is designed to see the consequence of a failed remote call. It also implements the *HelloWorld* interface. However, once called, the method *sayHi()* will suspend for some time before it returns.

The main purpose for this test case is as our implementation provides the option to set up a timeout while calling the method.

HelloInconsistency

The *HelloInconsistency* is designed to see how a failed remote call could result in inconsistency between server and clients. It also implements the *HelloWorld* interface. Both the remote object and the local client have a counter. Once called, the method *sayHi()* will add the counter by 1 and return the counter as part of the return value. Each time the local client successfully calls the *sayHi()*, it will add the local counter by 1.

To simulate the possibility of a failure, the method *sayHi()* might suspend for some time before it returns, resulting a failed call in the client. In this situation, inconsistency happens as the server adds the counter by 1 while the client fails to do so.

Further Works

If we've got enough time, we would like to add the following features and optimizations to our framework:

- *Merge RMIRegistry and RMIProxyDispatch* This is relatively a trivial task as we already have all the working codes and just need to merge them.
- *Support for Listing of Remote Objects* This is a little bit complicated as we need to add a new message type and might further consider how clients could utilize the list of remote objects.
- *Support for Downloading of Classes* This could be done on top of the listing feature. If a client could know all the remote objects and their classes, it could ask the server to download some classes it does not have locally. A possible challenge for this task is how to implement or use the HTTP protocol.