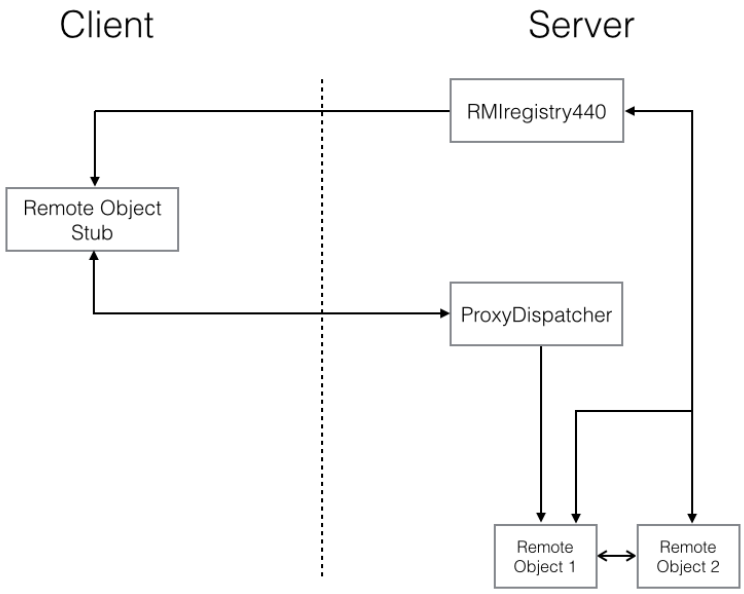
Report

Course: 15640 Distributed Systems
Lab: Lab2
Students: Jeremy Fu (Andrew ID: gengf)
Qing Wei (Andrew ID: qwei)

Design

In this lab, we emulated the behavior of the java.rmi package. In general, we tried to allow objects in one Java Virtual Machine (JVM) to be invoked from another JVM.



Application Programming Interface

We followed the java.rmi program paradigm to design our packages. From an application programmer's perspective, the remote object needs to be instantiated, exported and rebound in order to make it invoked remotely.

A remote interface which extends the Remote440 interface is required. The remote interface defines the methods that could be invoked remotely. Then the implementation class needs to implement the remote interface. The implementation of classes could defines both local and remote methods but only remote methods will be exposed to be invoked remotely. After an object is instantiated from a class which implements remote interface, it needs to be exported and obtains the corresponding stub. It is worth to mention that a remote object is not necessary to be rebound on RMI registry because some remote objects can be instantiated by other remote objects and the stub can be passed via return value, which also means that the stub won't necessarily be obtained via RMI registry.

In our design, once a remote object is exported, it is registered on a proxy dispatcher that listens for incoming invocations and distributes the invocations to different exported remote objects on the given listening port. And the stub also implements the same interface as remote object implements which ensures the client can call such methods remotely. But as a stub, it doesn't perform the function. Instead, it marshalls the parameters, sends the invocation to the remote host and unmarshalls the return value.

Our design is different from the paradigm provided in Honda's and we choose to emulate how Java.rmi packages provide to application programmers. This is because under Java's paradigm, it provides more flexibility. For example, application programmers can decide whether and/or when to export a remote object. And it is more flexible to manage the remote objects as well.

Components

RMIregistry

RMIregistry is a server. It maps the service name to the stub of remote object. A stub of RMIregistry is instantiated when called by static methods of LocateRegistry440 along with RMI registry host and port. The RMIregistry stub communicates with RMIregistry server to get results of requests back. The RMIregistry server accepts and handles request both from server and client side. It accepts five kinds of request:

- **Handshake:** Handshake request is used to detect the availability of RMIregistry in LocateRegistry440 class. It is reserved for further implementation of security as well. For example, a server could restrict the access availability to a set of hosts and deny requests from other hosts. Distinguished from the design of Java's LocateRegistry, where a RMIregistry stub is instantiated while no communication occurs, in our design, the getRegistry() of LocateRegistry440 sends the Handshake message in the process of instantiating the RMIregistry stub. Some exceptions may be thrown in situations such as host is not reachable, security policy restricts the access. We believe it is more secure to check the availability of RMI registry before sends messages with essential parameters.
- **Rebind:** Rebind request is to rebind the service name to the stub on RMIregistry. If the service name has been used, the corresponding stub will also be updated without warning. We leave the responsibility to users that naming service globally. The philosophy of this

design is based on that RMIregistry is running on the server side and we leave the flexibility to application programmers to handle such situation.

- **Unbind:** Unbind request is used to inform the RMIregistry server to delete the corresponding service name and stub information that had been registered on. Once a service is unbound, the stub of it will not be available on RMIregistry anymore. If the service requested is not available on registry, a NotBoundException will be thrown.
- **List:** List request gets back all the services names registered on RMIregistry currently.
- **Lookup:** A Lookup request looks for a specific service on RMIregistry server. The RMIregistry will then return the associate stub of that service back to client. If the service requested is not available on registry, a NotBoundException will be thrown.

Messages

We define the message to be the abstraction of commutation through network in our RMI framework. Considering our RMI framework, there are network communication between Remote Object Stub and ProxyDispatcher, Remote Object Stub and RMIregistry server, ProxyDispatcher and RMIregistry server. We introduce different types of message classes to represent different communication between each part. All specific messages inherits from the Message class, which defines the basic behaviors of different messages between RMI components in the following way:

- **Message Type:** Message type includes QUERY and REPLY. When a component sends request, the type of the message it sends is set to QUERY. The return message of that request is set to REPLY by the components handling the request.
- **Message Code:** Message code includes OKAY, DENY and EXCEPTION. It serves to mark the state of return message of a corresponding request. When the request is allowed and successfully handled, the code of return message is set to OKAY. When one RMI component finds the request is not allowed or unknown, the return message code is set to DENY, indicating the required operation cannot be handled by the receiver. When an exception is thrown by the invoked method, EXCEPTION is set as the state of the return message so that the stub on client side can be notified by this code and thrown corresponding exception that returns.
- **Message Operation(Op):** Message operation includes HANDSHAKE, REBIND, LOOKUP, UNBIND, LIST, INVOKE. Since the requests between each RMI components are strictly defined, the operation state in a message clearly informs the receiver the operation it needs to do. When a message is arrived on RMIregistry server or ProxyDispatcher, it checks the MessageOp of the incoming message and cast the message to corresponding subclasses and function, otherwise it refuses to handle the message.

Different kinds of messages are introduced to represent communication between components. which all inherit from the Message class:

- **Handshake Message:** Handshake message is used to send handshake message to RMIregistry server in the process of instantiating a registry stub.

- **Invoke Message:** Invoke message stores all the necessary information for a remote method invocation. When a remote method is invoked on the remote object's stub, the stub marshalls all the parameters and method name into a invoke message and sends to ProxyDispatcher.
- **Ret Message:** Ret message plays as the return message for a remote method invocation. It holds the return value as an object if the method's return type is not void. It also stores the specific exception thrown in the method call.
- **List Message:** List message plays as the request and return message to handle a LIST request to RMIregistry server.
- **LookUp Message:** LookUp message is used as the request and return message to handle a LOOKUP request to a RMIregistry server.
- **Rebind Message:** Rebind message is used as the request and return message to handle a REBIND request from a server to a RMIregistry server.
- **Unbind Message:** Unbind message is used as the request and return message to handle a UNBIND request from a server to a RMIregistry server.

Remote Object

A complete remote objects class consist of 3 major components: (1) remote interfaces, (2) classes implement interfaces declared in (1) and perhaps additional methods, (3) stub classes that implement the same interfaces as (2).

Here we take the SayHello(/src/example) as an example to illustrate it.

- **SayHelloInterface.java:** This file defines the remote interface. In order to make the remote interface recognizable by the system, the remote interface must extend the Remote440 interface. As mentioned above, the remote interfaces define the methods exposed to clients. Here we have sayHello() and createPerson() methods.
- **SayHello.java:** This file is the implementation of SayHello class. Application programmers need to declare and implement all the methods declared in SayHelloInterface.java. Here we have implemented sayHello() and createPerson() which will run on server. The implementation is the same as the implementation of objects running locally, which makes the remote invocation transparent to application programmers. As shown in this example, a local method beHappy() is implemented as well. This method won't be exposed to clients and it might perform as helper functions.
- **SayHelloInterface_Stub.java:** This file is the stub class for remote objects. In commercial package, this class is automatically generated and compiled. Due to the time limit, we don't implement the stub complier. However, we provide very clear paradigm and easy helper function to help build the stub. The SayHelloInterface_Stub class has to implement the SayHelloInterface interface, because SayHelloInterface_Stub needs to have the same remote methods exposed to the clients. Besides, it also extends the Stub440 class that defines the what to be stored in the stub in order to invoke remotely and the helper methods to invoke remote methods. The methods in the _stub class generally marshalls the parameter and use the invokeMethod() helper method to send the invocation request and receive the response.

Remote object stub

A remote object's stub acts as remote object's proxy to communicate with the host that the remote object resides on. A stub implements the same interface as the remote object does, it also extends Stub440 class, which provides a general function to communicate with the remote object's host. The bodies of methods in a stud does not do what a real remote object does, it just marshalls all the necessary information for a remote method call into an object of InvokeMessage, sends it to the corresponding proxy dispatcher, gets the return message back, unmarshalls the message and returns required information.

Proxy Dispatcher

At least one proxy dispatcher is instituted the first a remote object is exported. If the user set the port number to zero, the system will randomly choose a port number for the dispatcher. Each dispatcher holds references to one or more remote objects but each remote object can only be managed by one dispatcher.

In the situation where a remote object may be held by several different clients simultaneously, the timing to invoke the method on the same remote object needs to synchronized. Java has achieved this by synchronize the invocation internally. In our design, we achieve this by executing the invocation on same dispatcher sequentially. We sacrifice the efficiency to avoid race conditions. Another way to achieve this is to require users to declare synchronized on each methods in remote interface. But this way is not reliable.

The performance suffers more if several remote objects are all attached to such dispatcher. As a result, users can specify a port number for the dispatcher and bind the remote object to that dispatcher. Therefore such dispatcher is for one remote object exclusively and the response will be faster.

Naming

The remote object is named by Naming.java. The naming service serves two functions in general. First, it generates an object key and constructs the remote object reference for the stub. Second, it stores the remote object and its reference pair. The names for remote objects are unique within one JVM.

Pass by value vs Pass by reference

In our package, we strictly follow the way that the java.rmi package deals with the parameter and return value passing: all remote objects and has been exported are passed by reference. All remote objects that haven't been exported, local objects that implements serializable are passed by reference.

When the invokeMethod() is called in the stub, the parameters is checked by the above rules. Similarly, when the return value is checked before returned in proxy dispatcher.

Exception

Our framework also handled several common exceptions in RMI with our customized exception class.

- **RemoteException440:** A super class for all the exceptions related to communication through a remote method call.
- **ServerException440:** A ServerException440 exception is thrown when an error occurs on server side during the process of remote method invocation, e.g., connection between stub and remote object server fails.
- **NotBoundException440:** If the registry cannot find the specific record associate with the name of the service in the process of lookup or unbind, this exception will be thrown.
- **RegistryNotFoundException:** A RegistryNotFoundException will be thrown when an error occurs on the RMI registry server, e.g., connection cannot be made with registry server.

Unimplemented

Garbage Collector

Due to the limits of time and resources, we are unable to collect the remote object once it is no longer referenced remotely. In order to achieve this, we need a counter to track how many references both remotely and locally pointing to the remote object. And Java accomplishes this with JVM. Due to the fact that we are unable to keep track of the number of references pointing to the object, program which runs a remote object will not end, which leads to terminating the program manually by ctrl+c.

Parameter Check

Another part we haven't solved completely is the parameter check. We provide an elementary way to check the parameter: If the parameters themselves or return value are not collections, we can find the exported remote objects and substitute its stub. Java provides a complete solutions by overwriting the writeObject() method of ObjectOutputStream. The new methods replace all exported remote objects before write to the socket. Again this requires the low level support of Java.

Examples

We provided 3 examples in our codes: SayHello, Person and Compute[1]. First of the two examples are used to test the correctness and robustness of our program. The Compute is based on Java RMI tutorial to provide more flexibility to test our codes.

SayHello: SayHello class consists of 4 methods, 3 of which support to be invoked methods and beHappy() method is a local methods because it is not declared in its remote interface.
It is important to mention that createPerson() method creates, exports and rebinds a Person remote object. And a Person stub is returned. The sayHello() method throws a user-defined exception if the person object as its parameters is named "Kim", which is to test the correctness of catching checked exception and passing the exception to the client side.

Person: Person class works with SayHello class jointly. It is used to test the correctness of passing primitive variables and objects.

Compute: Compute class is a classic example in Java RMI. It works like a computation proxy accepting computing task which implements remote interface Task. We provide a Pi class to calculate the a designate precision of Pi.

Example demo
1. test1Client:
Purpose: This program tests correctness of passing parameter and return value of remote methods when they are both primitive types.

Step 1: Start RMIregistry on host A
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on host A.
$ java -cp RMI440.jar test.testPersonServer <put your service name here>

Step 3: Start client to get a person stub, set its name to 22 and get its name on host B.
$ java -cp RMI440.jar test.test1Client <service name on RMIregistry> <registry IP> <registry port number>

2. test2Client:
Purpose: This program tests correctness of passing parameter and return value of remote methods when they are both non-remote serializable object.

Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on host A.
$ java -cp RMI440.jar test.testPersonServer <put your service name here>

Step 3: Start the test2Client on host B.
$ java -cp RMI440.jar test.test2Client <service name on RMIregistry> <registry IP> <registry port number>

3. test3Client:
Purpose: This program tests the case in which the remote method's return value is an exported remote object (Pass by reference).

Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on RMIregistry on host A.
$ java -cp RMI440.jar test.testSayHelloServer <put your service name here>

Step 3: Start the test3Client on host B.
$ java -cp RMI440.jar test.test3Client <service name on RMIregistry> <registry IP> <registry port number>

4. test4Client (this one is a little bit different from others):
Purpose: This program tests two cases: A.the passed-in parameter in the remote method that hasn't been exported; b.the passed-in parameter is an exported remote object.

Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on RMIregistry on host A.
$ java -cp RMI440.jar test.testSayHelloServer <put your service name here>

Step 3: Start the RMI registry on host B.
$ java -cp RMI440.jar application.runRegistry <registry port number> &
note: By default, the registry port number is 1099.

Step 4: Start the test4Client on host B.
$ java -cp RMI440.jar test.test4Client <service name on RMIregistry> <registry IP> <registry port number>

Note: The following is the result of this test.

Person's name on server side: DEFAULT
Person's name on client side: Chris
/*------Now exported person object--------*/
Person's name on server side: DEFAULT
Person's name on client side: DEFAULT

In this program, a Person object is instantiated but not exported at first. The person object is passed to host A where the person's name is reset to "DEFAULT" and the return the name value seen by host A. As it shows "DEFAULT", the person's name seen by server is "DEFAULT". However, when we call getName() method on host B, it still shows the stale name "Chris".

However, when this object is exported later, the same operations are conducted, however both host A and host B see its new name as "DEFAULT". This demonstrates the correctness of the way to pass the parameters as we illustrated previously.

5. test5Client:
Purpose: This program tests the case in which a checked exception is thrown by the remote method.

Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on RMIregistry on host A.
$ java -cp RMI440.jar test.testSayHelloServer <put your service name here>

Step 3: Start the test5Client on host B.
$ java -cp RMI440.jar test.test5Client <service name on RMIregistry> <registry IP> <registry port number>

Note: The following is the result of this test.

```
Hello, Andy
java.lang.Exception: Found Kim.
        at example.sayhello.SayHello.sayHello(SayHello.java:18)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at util.ProxyDispatcher.run(ProxyDispatcher.java:72)
        at java.lang.Thread.run(Thread.java:744)
```

As it shows, the first time to call sayHello() method remotely the Person object is named with Andy. As a result, it returns the String “Hello, Andy”. However, in the second, we set the name to “Kim” which leads an exception thrown by SayHello object in the server. And the above results show we successfully catch the result. Moreover, we delete the some irrelevant exceptions in communication module, so that the exception happens as it it occurs on the same machine. In other words, it makes transparent to users.

6. test6Client:
Purpose: This program tests the case in which a runtime exception (NullPoinerException) happens in the remote method call.

Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start a server to export and rebind a SayHello remote object on RMIregistry on host A.
$ java -cp RMI440.jar test.testSayHelloServer <put your service name here>

Step 3: Start the test6Client on host B.
$ java -cp RMI440.jar test.test6Client <service name on RMIregistry> <registry IP> <registry port number>

Note: The following is the result of this test.

```
java.lang.NullPointerException
        at example.sayhello.SayHello.resetPerson(SayHello.java:49)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at util.ProxyDispatcher.run(ProxyDispatcher.java:72)
        at java.lang.Thread.run(Thread.java:744)
```

Since the test6Client program calls the SayHello remote object's resetPerson method and takes a NULL as parameter passed in. The remote object's host receives the remote request and calls the resetPerson method through its SayHello object, which throws a NullPointerException because the resetPerson method expects a object of Person so as to call its setName method. The runtime exception is caught by our RMI communication module and the client side can then get the exception, just as a local method call does.

7. Compute Pi:
Step 1: Start the RMI registry on host A.
$ java -cp RMI440.jar application.runRegistry &
note: By default, the registry port number is 1099.

Step 2: Start the ComputeEngine to export and rebind a Compute remote object on host A.
$ java -cp RMI440.jar application.ComputeEngine <put your service name here>

Step 3: Start the ComputeEngineClient on host B
$ java -cp RMI440.jar application.ComputeEngineClient <Precision of pi> <registry IP> <registry port number>

Reference

[1] http://docs.oracle.com/javase/tutorial/rmi/