15440 Lab2 Report — RMI Facility

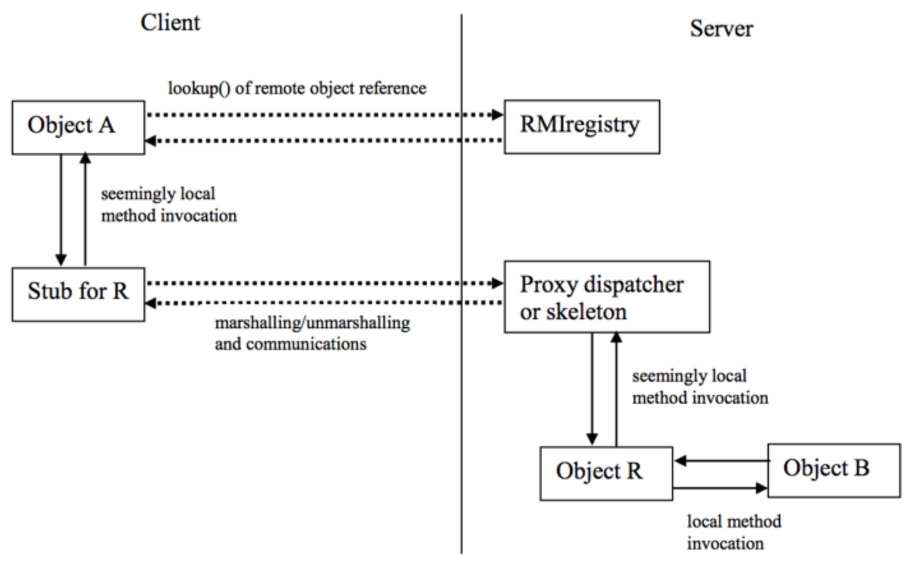
Wenting Shi(wentings)
Yijie Ma(yijiem)

Introduction:

In this project, we have been asked to implement a Remote Method Invocation facility in Java without using anything from the Java RMI package. The basic requirement for the facility is the ability to name remote objects, 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) and the ability to locate remote objects, e.g. a registry service. Before approaching this lab, we carefully studied Java's built-in RMI facility and the interface it provides for application programmers. We very much appreciate the Java's approach and therefore in a lot of ways try to implement the RMI in a similar neat and beautiful way. We have achieved all the basic requirements and have a fully-functional facility. Yet out approach has some limitations and does not include a stub compiler. Therefore, we provide hand-written examples and stubs for testing purpose.

Design Consideration:

Generally we followed the suggested design approach as indicated by the following diagram:


Client
Server
lookup() of remote object reference
Object A
RMIregistry
seemingly local method invocation
Stub for R
Proxy dispatcher or skeleton
marshalling/unmarshalling and communications
seemingly local method invocation
Object R
Object B
local method invocation

Before we sat down and start to implement the individual components, we decided to figure out what interface we would like to provide to the application programmers who wants to use our facility. We opted for a similar approach as Java's own RMI facility. Specifically, we have decided to use the following simple interface:

- Users (application programmers) who would like to use this RMI facility will need to provide three components: (1) an interface (remote interface) definition which describes all the remote methods the user would like to invoke on a remote object. This interface must extends our **Remote440** interface which contains no method definition but allows us to identify remote object from non-remote objects (2) an Implementation of this remote interface which contains a main method to bind this object to a service name to the registry. (3) a client program which uses the RMI facility.

- As a very first step, user will need to launch RMIRegistry, which provides mapping between service names and remote object references.

- User's implementation class can bind service name with remote object through static function we provide: **Naming440.rebind(String, Object).**

- User's client program can obtain a stub by calling another static function we provide as: **Naming440.lookup(String).**

- Once user's client program successfully obtains this stub, it will be able to invoke methods as if they are local procedures calls

Once we have this interface in mind, we began to think about how we can implement the system to achieve this simple and neat interface. The following are some key aspects of our design:

- We use a RMIMessage class to represent the communication incurred by RMI, it makes things easier to manage. It carries the remote method name, argument list, remote object reference, as well as exception log, return object.

- A RemoteObjectReference class is used to better represent the reference information, it contains the location of the remote object(host, port number) as well as the interface name. Most importantly it contains a unique objectID.

- RMIRegistryServer class manages the mapping between service names and remote object references, it listens on a specified port for "lookup" and "rebind" requests.

- A dispatcher class RMI440 is used to listen for Remote Method Invocation request on a separate port, this dispatcher is launched in a separate thread when user launches the RegistryServer. The dispatcher class is responsible for communicating with client-side stubs and the marshalling/unmarshalling of RMIMessages. It maintains a mapping between remote object references and the local object references.

- A client-side stub class is used to handle the marshalling of the method invocation into a message, delivery of the message to the dispatcher, and the reverse of his process, all the way to the client object, upon the methods return. Ideally, the stub is auto-generated by a stub compiler. Here we use hand-written stubs. The stub class implements the interface to provide application programmer an LPC illusion.

- If at any point of the invocation, a failure occurs, we will throw a RemoteException440 to indicate the failure.

Once we made the simplest case of Remote Method Invocation to work, we considered the case where either method parameter or return type contains Remote Object References, we handled this by using the following approach:

- Upon receiving the arguments contained in a RMIMessage, the dispatcher inspects each element of the argument list. Once it detects a remote reference (more accurately a instance of a stub object), it use the unique object ID to replace it with the real local object

- Upon a remotely-invoked method returns, the dispatcher once again inspects the return value. If its an instance of a remote object (which extends our required Remote440 signature), it returns the client with a stub instead of the remote object itself.

Testing:

We have provided two testing programs (each including a remote interface, an implementation and a client side test).

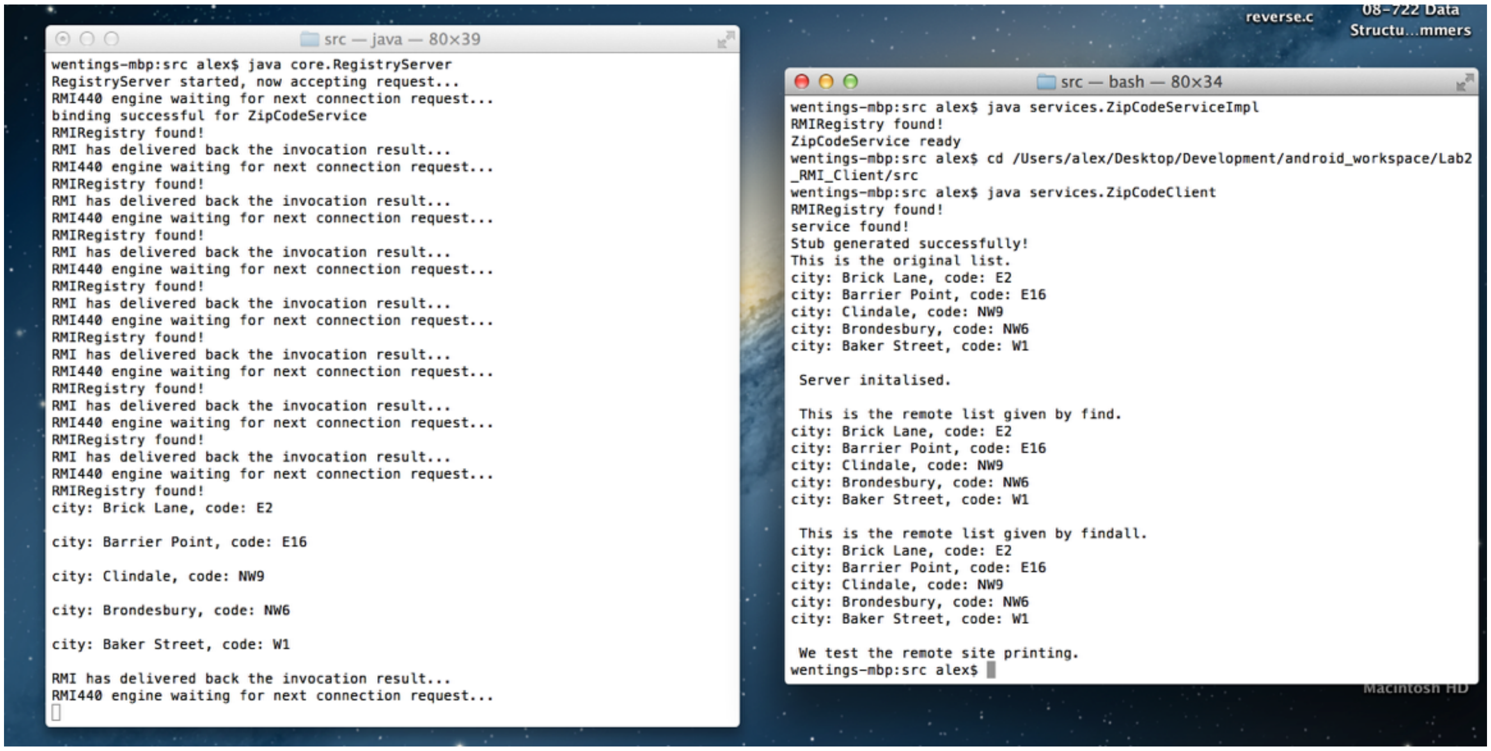
(The following is targeted for testing on a single machine, testing on separate machines are very similar)

In order to test the program, first compile everything in the src folder of both the Lab2_RMI and Lab2_RMI_Client project.
- Navigate to Lab2_RMI/src/core and type: **javac *.java**
- Navigate to Lab2_RMI/src/services and type: **javac -cp .. *.java**
- Navigate to Lab2_RMI_Client/src/core and type: **javac *.java**
- Navigate to Lab2_RMI_Client/src/services and type: **javac -cp .. *.java**

The first test program uses a local file to construct a list of Cities and ZipCodes, then manipulate the list using remote method invocation. To test the first program, follow the following step:

- Navigate to Lab2_RMI/src and start RMIRegistry: **java core.RegistryServer.java**
- Open another terminal window
- Navigate to Lab2_RMI/src and binds the service using the main function of the implementation class: **java services.ZipCodeServiceImpl**
- Navigate to Lab2_RMI_Client/src and start the client program: **java services.ZipCodeClient**

Upon successful invocation the following will be displayed in the console:



```
wentings-mbp:src alex$ java core.RegistryServer
RegistryServer started, now accepting request...
RMI440 engine waiting for next connection request...
binding successful for ZipCodeService
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
city: Brick Lane, code: E2

city: Barrier Point, code: E16

city: Clindale, code: NW9

city: Brondesbury, code: NW6

city: Baker Street, code: W1

RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
```

```
wentings-mbp:src alex$ java services.ZipCodeServiceImpl
RMIRegistry found!
ZipCodeService ready
wentings-mbp:src alex$ cd /Users/alex/Desktop/Development/android_workspace/Lab2
_RMI_Client/src
wentings-mbp:src alex$ java services.ZipCodeClient
RMIRegistry found!
service found!
Stub generated successfully!
This is the original list.
city: Brick Lane, code: E2
city: Barrier Point, code: E16
city: Clindale, code: NW9
city: Brondesbury, code: NW6
city: Baker Street, code: W1

 Server initalised.

 This is the remote list given by find.
city: Brick Lane, code: E2
city: Barrier Point, code: E16
city: Clindale, code: NW9
city: Brondesbury, code: NW6
city: Baker Street, code: W1

 This is the remote list given by findall.
city: Brick Lane, code: E2
city: Barrier Point, code: E16
city: Clindale, code: NW9
city: Brondesbury, code: NW6
city: Baker Street, code: W1

 We test the remote site printing.
wentings-mbp:src alex$
```

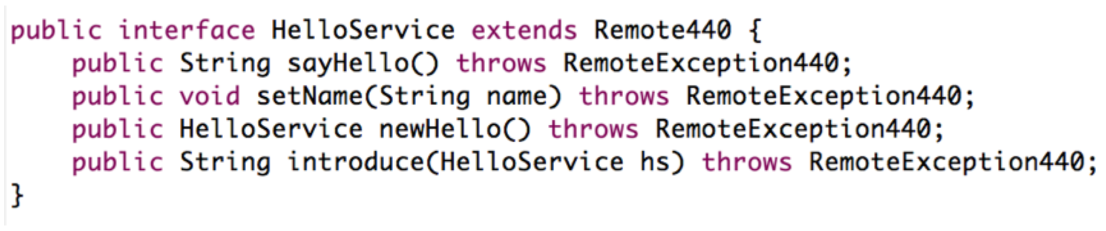
The screenshot on the right shows the result of client program using remote method invocation to perform operation (**find, findAll**) on the data. The screenshot is the test of the **printAll** method, which prints the list in the remote site(server).

The second test program mainly tests the handling of remote method invocation with Remote Object Reference as argument or return type.

To test the second program, following the following step:

- Navigate to Lab2_RMI/src and start RMIRegistry: **java core.RegistryServer.java**
- Open another terminal window
- Navigate to Lab2_RMI/src and binds the service using the main function of the implementation class: **java services.Hello**
- Navigate to Lab2_RMI_Client/src and start the client program: **java services.HelloClient**

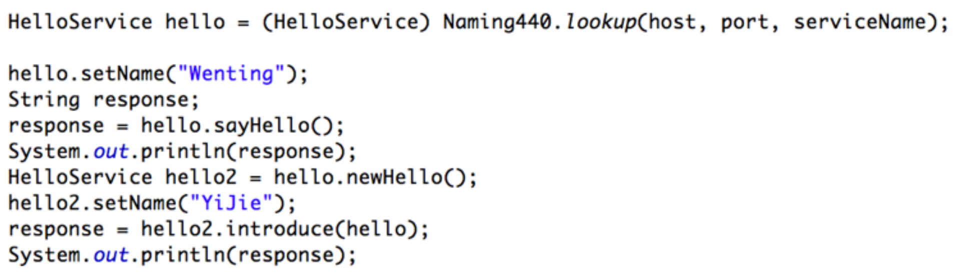
The remote interface contains the following method:



```java
public interface HelloService extends Remote440 {
    public String sayHello() throws RemoteException440;
    public void setName(String name) throws RemoteException440;
    public HelloService newHello() throws RemoteException440;
    public String introduce(HelloService hs) throws RemoteException440;
}
```

Each Hello remote object contains an instance variable name, **sayHello()** uses this name to return a string, **setName(String name)** sets the name, **newHello()** return a new remote reference, and **introduce(HelloService hs)** uses combines the two names together and returns a string.

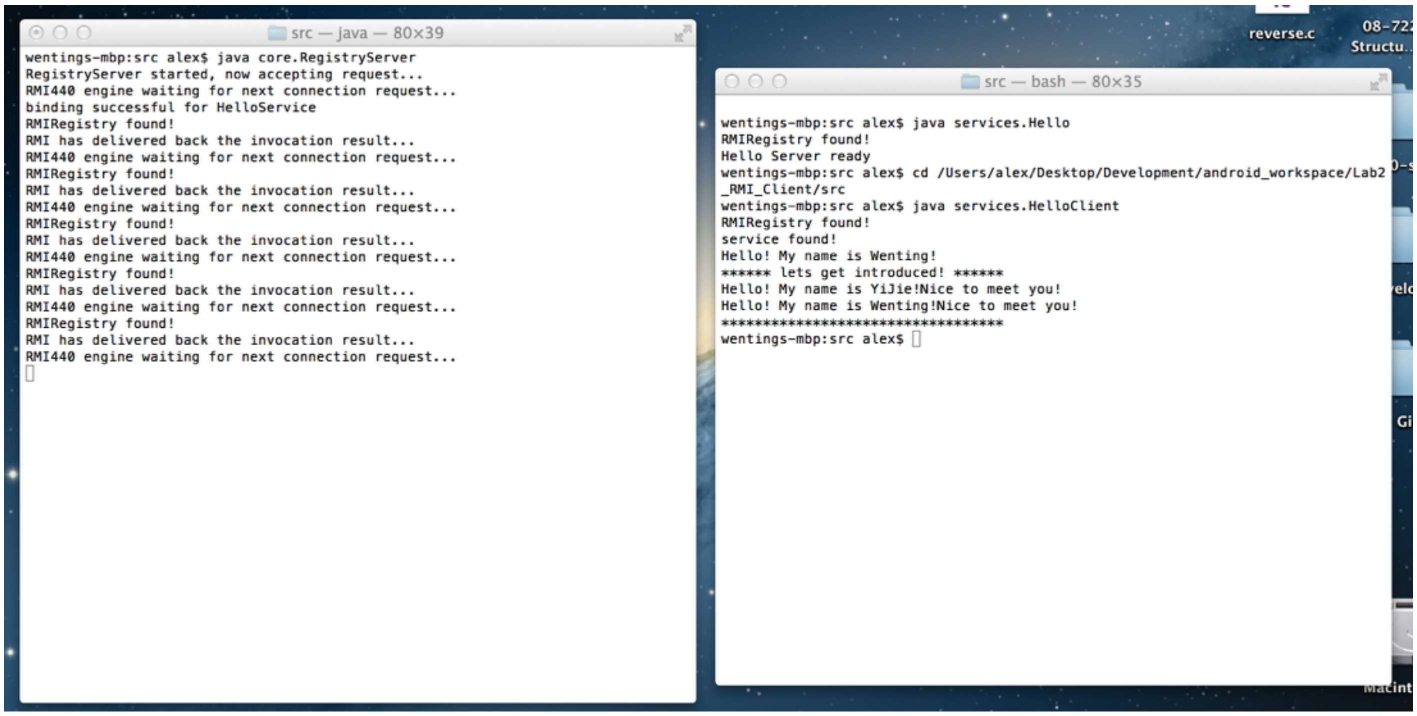
The following is the relevant client code:



```java
HelloService hello = (HelloService) Naming440.lookup(host, port, serviceName);

hello.setName("Wenting");
String response;
response = hello.sayHello();
System.out.println(response);
HelloService hello2 = hello.newHello();
hello2.setName("YiJie");
response = hello2.introduce(hello);
System.out.println(response);
```

Upon successful execution, the following output should be expected:



```
wentings-mbp:src alex$ java core.RegistryServer
RegistryServer started, now accepting request...
RMI440 engine waiting for next connection request...
binding successful for HelloService
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
RMIRegistry found!
RMI has delivered back the invocation result...
RMI440 engine waiting for next connection request...
```

```
wentings-mbp:src alex$ java services.Hello
RMIRegistry found!
Hello Server ready
wentings-mbp:src alex$ cd /Users/alex/Desktop/Development/android_workspace/Lab2
_RMI_Client/src
wentings-mbp:src alex$ java services.HelloClient
RMIRegistry found!
service found!
Hello! My name is Wenting!
****** lets get introduced! ******
Hello! My name is YiJie!Nice to meet you!
Hello! My name is Wenting!Nice to meet you!
**********************************
wentings-mbp:src alex$
```

Limitations:

1. Our solution right now lacks a automatic stub compiler and requires hand-written stubs.
2. When the client does not have a required class file, our solution currently does not support a HTTP connection to the server to retrieve it
3. Our design listens on two port and have some communication overhead between these two ports (between the dispatcher and the registry server)