Swapnil Pimpale (spimpale)
Romit Kudtarkar (rkudtark)

15-440/640 Project 2 Report

Collaboration Statement: All work submitted by us is our own.

Introduction
This report contains information about the design and implementation of RMI framework that we have written for Project 2.

Overview of Design
Our framework has the following main components - *Registry, Stubs, Remote Object References (RORs), Dispatcher/Server and the Clients*.

The registry is a collection of remote object references. The clients query the registry using servicenames and get back RORs. The RORs are used to construct stubs on the client side. This is called localisation. The stubs in our case are hand-written. The clients then invokes methods on the stub. The stub marshals the method invocation request and sends it to the dispatcher over a socket connection. The dispatcher then unmarshals the request, invokes the appropriate method on the local object, marshals the result and sends it back to the stub. The stub then unmarshals this response and sends the final result to the client.

Marshalling
Our framework utilizes marshalling to ensure smooth communication between the Client and the Dispatcher.

For any method invoked on the client's stub, the method parameters are marshalled accordingly:

- Parameters that do not implement the **RemoteInterface** interface are serialized and sent over a socket connection (pass-by-value)
- Parameters that implement the **RemoteInterface** interface have their corresponding ROR serialized and sent over the socket connection (pass-by-reference)

On the Dispatcher side, the method parameters are read in from the socket and unmarshalled accordingly:

- Parameters that are not RORs are left untouched

- Parameters that are RORs have their corresponding local objects looked up and used in place of the ROR for the method invocation

When the Dispatcher attempts to return a value back to the stub, the following marshalling occurs:

- If the return value does not implement the **RemoteInterface** interface, the value is serialized and sent over a socket connection (pass-by-value)
- If the return value implements the **RemoteInterface** interface, the value's corresponding ROR is looked up and sent over the socket connection (pass-by-reference)

When the stub receives a response from the dispatcher, it performs the following unmarshalling:

- If the return value is not a ROR, the value is simply passed back to the client
- If the return value is an ROR, the stub for the ROR's corresponding object is constructed using the localise method. This stub is then passed back to the client.

Implementation

Registry:

- The registry is maintained as a hash table which maps a "servicename" to an ROR.
- The registry has an IP Address and a port number associated with it.
- It implements **RegistryOperationsIntf** interface (*lookup(), rebind(), listAllROR(), getSize()*)
- At the start, the registry creates a server socket using the IP Address and port combination and waits in a continuous loop for requests.

Stubs:

- The stubs are handwritten and have the same method signatures as the class that they represent.
- The stubs perform the following tasks:
    - Create a socket connection to the ROR's location
    - Marshal the parameters
    - Create an **RMIMessage** with the methodName, list of parameters and their types and the **objKey (unique key)** for the ROR
    - Send this message on the socket

    - Receive response on the socket and unmarshal it
    - Send the final response to the caller

Remote Object References

- The ROR basically consists of the *location information* about the real object
    - ipaddr: IP address of the server which has exported this object.
    - port: port number of the server which has exported this object.
    - remIntfName: name of the remote interface name implemented by this object.
    - objKey: Unique key associated with this ROR.
- If an exported remote object is passed as a parameter or return value in a remote method call, the ROR for that remote object is passed instead of the object itself.

Dispatcher/Server

- The dispatcher is the “server side” of the framework
- The dispatcher performs the following tasks:
    - Create local objects of the Class passed in from the command line
    - Locates the registry on the host and port passed in from the command line
    - Creates RORs for the objects and stores them in the registry
    - Create a server socket and enter a loop listening for requests from the clients
    - On receiving a request the following steps are performed:
        - The parameters in the RMIMessage are unmarshalled i.e., if an ROR is passed in we lookup for the corresponding local object in the ROR table.
        - We then use reflection to get the class for the local object. From the class we get it’s actual method.
        - We then execute **method.invoke()** with the local object and the unmarshalled parameters.
        - The result of the above invocation could be an exception, another remote object or some other value. Our framework can process all of this correctly.
        - We then store the **result** back in the RMIMessage and send it back to the stub. At this point the dispatcher is ready to serve the next request.

Clients

- The clients perform the tasks in the following order
    - Take in the registry IP and port from the command line

    - Locate if there is a registry present on that IP:port
    - Lookup the servicename and get an ROR
    - Localise a stub of the ROR
    - Invoke remote methods on the stub and get the result back

<u>Features of the Framework:</u>

- Registry and Dispatcher are run as separate programs, reducing contention time for client requests to be served. This way the dispatcher deals only with method invocation requests and the registry deals only with registry requests. This ability is good for scaling the framework for multiple clients, servers and registries.
- Our framework has the ability to correctly pass in and return RORs. The framework also supports passing in and returning of local unexported remote objects.
- The stubs are currently handwritten but a compiler for stub generation can be implemented with minimal efforts.
- Our framework has a separate message formats for **RegistryMessage** and **RMIMessage**.
- The communication module of the framework also supports setting and passing of exceptions.
- We can pass in the Classname, number of objects to be created and their servicenames to the dispatcher from the command line. This way the dispatcher is not restricted with just one type of Class objects. New remote objects and their implementations can be easily added to work with the framework.

<u>Assumptions:</u>

- A remote object implements a single Remote Interface.
- Remote objects passed in as parameters and returned from methods are all exported.

<u>Testcases</u>

To completely test the framework we have written the following test cases:

1. *Basic Test:*
    a. This test invokes **add(a, b), subtract(a, b), divide(a, b)** methods on a remote object. Note that 'a' and 'b' are primitives or non remote objects.
    b. If the value of b is 0, the server side code throws an **IllegalArgumentType** exception which is caught by our framework and returned back to the client.
    c. This test tests the basic RMI and exception handling functionality of the framework

2. *Remote Tests:*
    a. PassInExportedRO_Client: This testcase tests the working of the framework when remote objects are passed in as method parameters
    b. RetExportedRO_Client: This testcase tests the working of the framework when a remote object is returned from a method.

<u>Instructions for compiling and running our framework</u>

1) Copy all directories over from handin.
2) Run the following commands:

```
#> mkdir bin
#> javac -d bin comm/*.java dispatcher/*.java registry/*.java ror/*.java test/*.java
```

3) The registry and dispatcher can be run on the same or different machines. The command for running the registry is:

```
#> java -cp ./bin registry.ActualRegistryImpl <portNumber>
```

Port number is optional; if not provided default port number of 12346 will be used. This command will run the registry on the specified port. *Note that the registry must be started before starting the dispatcher.*

The command for running the dispatcher is:

```
#> java -cp ./bin dispatcher.Dispatcher <ServerPort> <RegistryHost>
<RegistryPort> <ClassName> <NumofObjects> <serviceNames…>
```

This command will start the dispatcher at <*ServerPort*>. The dispatcher will attempt to connect to the host and port specified by <*RegistryHost*> and <*RegistryPort*>, and will create <*NumofObjects*> instances of <*ClassName*> with the serviceNames specified by <*serviceNames…*>. Note that the <*ClassName*> to be constructed for the BasicTest_Client is different from the <*ClassName*> to be constructed for the PassInExportedRO_Client and RetExportedRO_Client.

There are three clients we have created: BasicTest_Client, PassInExportedRO_Client and RetExportedRO_Client.

The command for running the BasicTest_Client is:

#> java -cp ./bin test.BasicTest_Client ***<registryHost> <registryPort> <int1> <int2>***

The commands for running the other 2 clients are:

#> java -cp ./bin test.*<clientName> <registryHost> <registryPort>*

<u>Example run of all tests:</u>

On angelshark.ics.cs.cmu.edu
#> java -cp ./bin registry.ActualRegistryImpl

On bambooshark.ics.cs.cmu.edu
#> java -cp ./bin dispatcher.Dispatcher 12345 angelshark.ics.cs.cmu.edu 12346 test.testRMI 1 mathOperations

On makoshark.ics.cs.cmu.edu
#> java -cp ./bin test.BasicTest_Client angelshark.ics.cs.cmu.edu 12346 10 20
This will add, subtract and divide 10 and 20.

To run the remote tests, *kill the dispatcher on bambooshark and restart it* with the following command:
#> java -cp ./bin dispatcher.Dispatcher 12345 angelshark.ics.cs.cmu.edu 12346 test.RemoteTest 2 obj1 obj2

On makoshark.ics.cs.cmu.edu
#> java -cp ./bin test.PassInExportedRO_Client angelshark.ics.cs.cmu.edu 12346
#> java -cp ./bin test.RetExportedRO_Client angelshark.ics.cs.cmu.edu 12346

<u>Writing your own RemoteObjects</u>
To write your own test case you will have to do the following:
- Create an interface of methods (say MyTestInterface). MyTestInterface should extend RemoteInterface
- Write a class which implements this interface (say MyTestClass)
- Write a stub class on the client side (say MyTestInterface_stub). This stub class should extend Stub and implement RemoteInterface.

- Pass in the Classname, number of objects and their servicenames on the dispatcher’s command line.
- Write a client program which invokes these methods