Remote Method Invocation Framework
Final Report

Yiming Zong (Jimmy)
Carnegie Mellon University
Pittsburgh, PA 15213
yzong@cmu.edu

Abstract

This report reflects my final progress on Remote Method Invocation (RMI) Framework project for 15-440, Fall 2014. I will first present the functionalities supported by the framework, and then demonstrate the communication protocol between different components of the framework including RMI Registry, RMI Servers, and client applications. Also, I will include a Developer's Guide on developing RMI Applications based on my framework. Eventually, I will describe how to build and test the current framework from source, and survey some additional unimplemented features that would be desirable for a commercial package.

1 Functionalities Supported

1.1 Overview of Framework

The RMI Framework provides an abstract communication model between Java Virtual Machines (JVMs), such that local methods on Client Applications are able to interact with Remote Java Objects located on another JVM.

The Client Application, instead of storing the actual Remote Object, holds a *Remote Object Reference (RoR)* and a *Remote Object Stub* by querying the *RMI Registry*. The centralized RMI Registry keeps tracks of all *RoRs* by their unique service names, and each *RoR* contains the endpoint of the *RMI Server* that holds the *Remote Object* and the identifier of the Remote Object on that server. And, the Stub, which acts like an *Invocation Proxy*, handles all client-side method invocations by sending necessary information to the corresponding *RMI Server*, such that a local method invocation is made on the remote server and the result is returned to the Client Application via TCP socket.

The following sub-sections will survey the functionalities of each RMI Framework component – RMI Registry, RMI Servers, and Client Applications – without going into much details about the actual communication protocal, which will be covered in *Section Two*.

1.2 RMI Registry

RMI Registry holds the *RoR* for each Service (i.e. Remote Object) supported by the Framework. It supports the following operations from its clients (RMI Servers and Client Applications):

- `ping`: A client can test if the *RMI Registry* is alive and operational;
- `list`: A client can request a list of Service Names registered on the *RMI Registry*;
- `lookup`: A client (mostly *Client Application*) can query the *RMI Registry* with a *Service Name*, and ask for its corresponding *RoR*;

- `bind`: An *RMI Server*, upon loading an RMI Remote Object, needs to report (`bind`) to the *Registry* such that clients can query it via the *Registry*. This will *not* overwrite an existing entry with the same *Service Name* on the *Registry*;
- `rebind`: Same as above, except that the operation will overwrite an existing entry with the same *Service Name* on the *Registry* if it exists;
- `unbind`: When an *RMI Server* wants to take down a Remote Object or shuts down, it would remove (`unbind`) the services from *RMI Registry* such that Client Applications will also know that they are no longer available.

1.3 RMI Servers

RMI Servers hold the actual RMI Remote Objects for the entries in RMI Registry. Whenever a Client Application attempts to invoke a method on the remote object, the request is relayed to the hosting RMI Server for execution. In order to allow *live patching* of the RMI Objects, my implementation of *RMI Server* includes a CLI, where a system administrator may `bind`/`rebind`/`unbind` a *Service* via the build-in command line without restarting the *RMI Server*. It supports the following *live* operations:

- `list`: Lists all Remote Objects currently registered on the local RMI Server;
- `bind ServiceName RemoteObjectClass`: Binds an object of `RemoteObjectClass` as `ServiceName` on the *RMI Registry*. Returns an error message if `ServiceName` is already occupied in the RMI Registry;
- `rebind ServiceName RemoteObjectClass`: Same as `bind`, except that it would overwrite the entry with `ServiceName` on RMI Registry if it exists;
- `unbind ObjectKey`: Unbinds the Remote Object with `ObjectKey` on *RMI Registry*;
- `exit/quit`: Gracefully terminates the RMI Server by first unbinding all Remote Objects registered on the local server from the RMI Registry.

1.4 Client Applications

Client Applications utilize RMI Registry and RMI Server APIs to make Remote Method Invocations. The Client Applications use the following methods to initialize:

- `LocateRMIRegistry.getRegistry(host, port)`: If the given endpoint is a valid *RMI Registry*, return the `RMIRegistryClient` object for the Registry;
- `RMIRegistryClient.lookup(serviceName)`: Looks up the RMI Registry for a specific *Service Name*. If found, return the `RemoteObjectRef` object for the Remote Object; otherwise, return `null`;
- `RemoteObjectRef.localize()`: Obtains the local object stub for the remote object. With proper casting, this object can be used to invoke remote methods.

1.5 Special Features

Following are speical features that the author thinks are useful for application programmers and are beyond the request of the project specification:

- *Command Line Utility in RMI Servers:* As mentioned in *Section 1.3*, it allows dynamic loading of Remote Objects, which means more flexibility and shorter service down times;
- *Elegamt Error Catching:* RMI is prone to various Exceptions including network errors and invalid remote-side objects provided by the application programmer. This should not cause the entire service to go down, and in my implementation the Exceptions are carefully caught and the RMI Framework is brought back up whenever possible;
- *RMI Server Clean-Up Command (Weak Garbage Collection):* Before a RMI Server exits, it informs the RMI Registry to remove all Service Names that it supports, such that Client Applications will not attempt using them in the future;

- *RMI Invocation Exception Class:* `RMIInvocationException` class is created for abstracting all exceptions happened during a Remote Method Invocation (similar to `java.rmi.RemoteException` in Java's implementation). This makes it easier for client-side application programmers to deal with the Exceptions;

2 Communication Protocol: RMI Registry & Other Components

By discussion in *Section One*, *RMI Registry* only holds *(Service Name, RoR)* pairs, and each *RoR* only contains the the following fields: host name and port number of RMI Server, Object Key for the Remote Object, and the Remote Interface Name of the object on RMI Server. Note that the data fields are merely integers and Strings, so `BufferedReader` and `PrintWriter` based on TCP socket would suffice. As described in *Section 1.2*, interactions between an *RMI Registry* and its clients (including *RMI Servers* and *Client Applications*) are only `bind`, `rebind`, `unbind`, and `list` requests, so my implementation abstracts the requests to methods of an `RMIRegistryClient` class. Following are the communication details of each method:

2.1 `ping` Request:

Intended to be used by both *RMI Server* and *Client Applications*.

Client sends in a request header `PING`, and the Registry should always respond with `PONG`.

2.2 `list` Request:

Intended to be used by *Client Applications*.

Client sends in a request header `LIST`, and the Registry should return `OK` on the first line, number of registered services on the second line, and one Service Name per line after that.

2.3 `lookup` Request:

Intended to be used by *Client Applications*.

Client sends in a request header `LOOKUP` followed by the desired Service Name on the second line. If the service is not found, Registry returns `NOTFOUND` to the client; otherwise, Registry returns `OK`, followed by the fields of corresponding RoR line by line, such that the client can reconstruct the RoR from them.

2.4 `bind` Request:

Intended to be used by *RMI Server*.

Client sends in a request header `BIND` followed by the fields of its Remote Object Reference line by line. If the operation succeeds, Registry returns `OK`; otherwise it returns `ERROR`.

2.5 `rebind` Request:

Intended to be used by *RMI Server*.

Same as `bind` Requests, except that the request header is `REBIND`.

2.6 `unbind` Request:

Intended to be used by *RMI Server*.

Client sends in a request header `UNBIND` followed by the desired Service Name on the second line. If the service is not found or error occurs while unbinding, Registry returns `ERROR` to the client; otherwise, Registry returns `OK`.

3 Communication Protocol: RMI Server & Client-Side Applications

Referring to *Section 1.4*, once the Client-Side Application has obtained the local stub of the Remote Object by contacting the RMI Registry and then calling `.localize()`, the stub would act as an *Invocation Proxy* and contact the *RMI Server* that holds the Remote Object directly upon any method invocations. Since the communication between Client-Side Applications and the RMI Server involes passing around parameters and return values (which can be Serializable Objects), my approach was to wrap this type of communication in an `RMIInvocationPkg` and use `Object-InputStream` and `ObjectOutputStream` to send objects over TCP socket. Here is what happens when a Client-Side Application invokes a remote method:

Step One: Client application invokes method on Stub Object

The client first initializes by calling `localize` method on a `RemoteObjectRef` object we obtained from RMI Registry. Then, the client application invokes the method on the object returned from `localize` with usual parameters, as if it were the actual object that the Client Application wishes to interact with.

Step Two: Stub method constructs `RMIInvocationPkg`

The stub method first pre-processes the parameters passed into the method (described below) and then constructs an `RMIInvocationPkg` Object with the method name, processed parameters list, and the Remote Object Reference corresponding to the Stub.

For parameter-preprocessing step, there are two cases: for *non-Remote* objects such as int, String, and other Serializable types, we can pass the object directly as is (*Pass by Value*). However, if a parameter is a *remote object*, we wish to pass the RoR corresponding to the remote object instead (*Pass by Reference*), such that RMI Server can recover the Remote Object from the RoR. (See `com.yzong.dsf14.RMIFramework.examples.CalculatorServer.secretMethod` method for example.)

Step Three: Stub method sends invocation request to RMI Server

During `localize` step, we made the local stub contain a copy of its RoR, so it can identify which host and port to connect with. After constructing the `RMIInvocationPkg` as in previous step, the stub sends it to the RMI Server. When RMI Server responds with a return value, the local stub casts the returned value and relays the result to the client application.

Step Four: RMI Server restores remote method and arguments

After receiving an `RMIInvocationPkg` object from Client Application, RMI Server first obtains the actual Remote Object by querying its local `RoRTable` with the given RoR as key. Also, RMI Server will get the method name along with a list of the parameters' classes (types) in order to get the actual method by using `getMethod`.

As in *Step Two*, the caveat is to convert the *RoR* parameters back to their corresponding remote objects by looking up in the local `RoRTable`.

Step Five: RMI Server invokes method locally and sends back return value

After the remote object and methods are located, the method is invoked with `java reflect`, and its return value is sent back to the Client Application via an `ObjectOutputStream`. This completes a full cycle of Remote Method Invocation.

4 Developer's Guide – Using RMI Framework

Since the RMI Framework is to be used by application developers, a pre-compiled Javadoc of the project can be found in the `doc/` directory under the project root. Users can simply open

doc/index.html and access the documentation of the entire code base including the API usage guides. The mechanism of RMI Framework has been explained in previous sections.

In order to fit a Java Object into the RMI Framework, say, FooBarServerImpl, application programmer should first create an interface FooBarServerIntf that extends RMIRemoteStub, and the interface should include all functions that are to be accessible by the Client Applications. Then, the developer should create an alternative implementation of the FooBarServerIntf interface, named FooBarServer_stub, which includes all public methods as in FooBarServerImpl, but each method in this case sends an RMIInvocationPkg (as specified in *Section Two*) to RMI Server and returns the result given back by RMI Server.

Special care should be taken when a parameter of a remote method is an instance of some other Remote Object. In this case, instead of passing the Remote Object (which are actually *Stubs* in local application) directly to the RMI Server, we pass its RoR instead. (See com.yzong.dsf14.RMIFramework.examples.CalculatorServer_stub:69 for an example.)

In general, application developers are strongly recommended to look at the sample applications in the Package com.yzong.dsf14.RMIFramework.examples, and follow the existing code. To build and run the Framework, see *Section Five*.

5 Dependencies, Building, and Testing

The yzong handin directory will contain two sub-directories, i.e. reports and RMIFramework. In the former directory you can find this report file, and in the latter directory is the clean source code for the project.

5.1 Dependencies

This project uses two external libraries, Apache Commons CLI and Apache Commons Lang. The former one is used in handling the command-line arguments when running the RMI Registry and RMI Servers, and the latter is used to determine if a given string is a valid integer. Their jar files are included in the lib/ directory of the Java project.

Also, to build the project from scratch without using IDE, an Ant Buildfile, build.xml is included in the root directory of the Java project. To use it, the machine needs to have Ant installed.

5.2 Building Instructions

To build the project from scratch, follow the steps below:

```
gkesden@ghc11 yzong$ cd RMIFramework
gkesden@ghc11 yzong/RMIFramework$ ls
    build.xml  doc  lib  src
gkesden@ghc11 yzong/RMIFramework$ ant clean build jars
    (Output omitted)
gkesden@ghc11 yzong/RMIFramework$ ls
    bin        lib                      RMIServer.jar         src
    build.xml  RMICalculatorClient.jar  RMITestRegistry.jar
    doc        RMIRegistry.jar          RMIZipCodeClient.jar
```

To clean the project directory, run the following command:

```
gkesden@ghc11 yzong/RMIFramework$ ant clean
    (Output omitted)
gkesden@ghc11 yzong/RMIFramework$ ls
    build.xml  doc  lib  src
```

5.3 Testing Instructions

The following test routine is designed to survey most functionalities of the RMI framework. Feel free to experiment with different commands in RMI Server terminal and also different client applications. The machines and ports in use in the sample are `ghc12:6060` (RMI Registry), `ghc15:41052` (RMI Server), and `ghc16` (client-side application). Other combinations will also work, but be sure to substitute with the correct parameters below.

```
# Starts up RMI Registry
gkesden@ghc12 yzong/RMIFramework$ java -jar RMIRegistry.jar -p 6060
        *** Stdout tails RMI Registry activity log ***

# Sanity check for RMI Registry (-ea flag is necessary!)
gkesden@ghc15 yzong/RMIFramework$ java -jar -ea RMITestRegistry.jar
Connecting to RMI Registry Server...
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Testing PING...
Testing BIND...
Testing LOOKUP
Testing REBIND
Testing LIST
Testing UNBIND
All tests passed!

# Start an RMI Server
gkesden@ghc15 yzong/RMIFramework$ java -jar RMIServer.jar -h ghc12.ghc.andrew.cmu.edu -p 6060
INFO -- RMI Server started at ghc15.ghc.andrew.cmu.edu:41052.
        Master RMI Registry at ghc12.ghc.andrew.cmu.edu:6060.
INFO -- Connection to RMI Server established. CLI started.
RMI Server @ 41052 > bind service1 com.yzong.dsf14.RMIFramework.examples.CalculatorServerImpl
Successfully registered service service1!

RMI Server @ 41052 > bind service9 com.yzong.dsf14.RMIFramework.examples.ZipCodeServerImpl
Successfully registered service service9!

RMI Server @ 41052 > list
Following are the entries of Remote Object Reference table on local RMI Server:
Object Key: 0
Remote Interface Name: com.yzong.dsf14.RMIFramework.examples.CalculatorServer
Object Key: 1
Remote Interface Name: com.yzong.dsf14.RMIFramework.examples.ZipCodeServer

# Test the RMI ZipCode Client
gkesden@ghc16 yzong/RMIFramework$ java -jar RMIZipCodeClient.jar
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Registry Service Name for ZipCodeServer: service9
Data File Path: src/com/yzong/dsf14/RMIFramework/examples/ZipCodeData.txt
        *** Expected test output ***

# Test the RMI Calculator Client
gkesden@ghc16 yzong/RMIFramework$ java -jar RMICalculatorClient.jar
Registry Hostname: ghc12.ghc.andrew.cmu.edu
Registry Port Number: 6060
Registry Service Name for CalculatorServer: service1
Setting the name of Calculator Object as: CalcFooBar
        *** Expected test output ***
Registry Service Name for ZipCodeServer: service9
Data File Path: src/com/yzong/dsf14/RMIFramework/examples/ZipCodeData.txt
Initializing the ZipCodeSever...
        *** Expected test output ***

# Exit RMI Server; Clean-up Services on RMI Registry
```

```
RMI Server @ 41052 > exit
Unbound service service1 from RMI Registry.
Unbound service service9 from RMI Registry.
Done. Goodbye!
```

6 Further Work & Enhancements

Due to time constraints, although my final product is fully functional, it still lacks many features that are desired in commercial packages (like `java.rmi.*`), and their implementation difficulties also vary. This section will mention a selected few and discuss the difficulty for implementing them.

6.1 Multi-Threaded RMI Communication Modules

Currently, all communications between RMI Registry, RMI Servers, and client applications are sequential, which means that the RMI Registry or Server needs to wait for current operation to complete before processing another request. This can be problematic in production environment especially when there are many client applications and the remote method calls take a long time to finish. This issue can be solved by spawning a separate thread for handling each request; however, it makes locking and synchronization more complex, as we want to ensure proper locking and avoid race conditions.

6.2 Stub Compiler and Stub Transfer Mechanism

According to previous discussion, the current RMI mehcanism relies on the fact that the client applications have access to the RMI Stub for each Remote Object beforehand. However, this is unnecessary and inconvenient because in production environment the RMI Server might want to support new services dynamically without restarting the entire framework. In this case, a Stub Compiler can be used to generate the Stub objects dynamically, and the client-side applications can download them via HTTP or other protocols. It can be achieved by using Java's dynamic object proxy library (`java.lang.reflect.Proxy`) along with a web server (e.g. Apache Tomcat).

6.3 Elegant RMI Exception Reporting

Although my implementation of RMI Framework defines an Exception class `RMIInvocationException`, it does not provide much useful information to the user as in whether the RMI Registry or Server is down or an actual Exception in Remote Object occured. This can fixed by tracing the remote object invocation more carefully and extend the `RMIInvocationException` class with more useful information.

6.4 Client-Side / Infrastructure-Side Separation and Access Controls

In the current implementation, the client-side application has full access to classes and methods used to interact with RMI Registry and RMI Servers. This poses a significant security issue becaues malicious clients are able to `rebind` another service to the RMI Registry, and this enables the attacker to eavesdrop all method invocations from client applications, which might contain sensitive information. This can fixed by separating client-side RMI methods from the server-side codebase as much as possible and giving clients read-only permissions to key entities like the RMI Registry.