15-440
Project #2: Design and Implementation of a RMI Facility for Java

Team:
Abhishek Bhowmick (abhowmi1)
Neil Rajesh Dhruva (ndhruva)

Date:
10/10/2014

Contents

Problem Definition

We are required to implement an RMI (Remote Method Invocation) facility for Java that allows objects from one JVM to invoke methods on an object in a remote JVM that can be accessed by a network. The deliverables are :

1. Ability to name remote objects (remote object references)
2. Ability to invoke methods on remote objects and also pass parameters and receive results from those methods
3. Ability to locate remote objects through a registry
4. Example applications for testing the RMI system
5. Ability to catch exceptions thrown by remote objects (Remote Exceptions)

Solution Overview

The solution overview discusses of our implementation of an RMI facility in Java that can be used by an application programmer to invoke methods remotely. The following diagram gives a high level overview of our design.

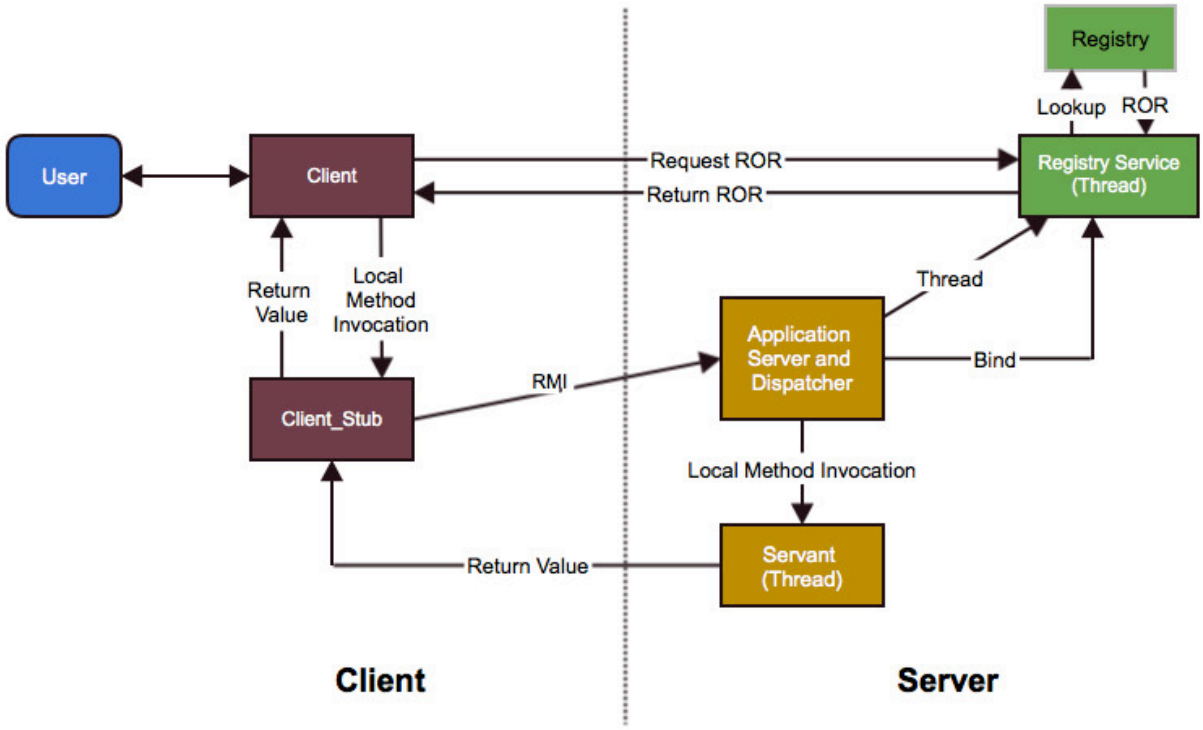

Figure: System Design

Much like the RMI facility provided by Java, our RMI implementation is able to give the illusion of local method invocation to the client, while the real work is done on the server. We assume that the server has downloaded the implementation class from the client. The client, instead of the actual implementation of the methods, has a stub that it communicates with locally. We also assume that this stub is available with the client (since we do not have a stub compiler for our RMI). The stub is instantiated using a remote object reference that the client obtains from a registry service. The stub in turn is responsible for communicating with the server, asking the server to execute a client method, and get the result from the server and pass it to the client. The functionality of the client and the server (including the registry) is made more clear below.

Server Side

We begin with the server side of the RMI, since we create the server first. In our implementation, the server side consists of the following components:

1. Application server and dispatcher:
   The application server is responsible for two things: creating a registry service, and listening to and doing preliminary processing on client requests. The application server instantiates a registry service that can hold remote object references. The service runs in its own thread and listens to client requests on a separate port. The application server then creates all the client implementations for which it already contains all the classes in its classpath (as we assumed earlier). For instance, if it has the implementation to the ZipCodeServer, then the server instantiates it. The server maintains a local table where it maps such an instantiation to a unique number. Additionally, it creates a Remote Object Reference (ROR) which refers to this ZipCodeServer object.
   Hence, our implementation of the ROR is slightly different from that in Java. We associate a separate ROR for all of client's remote classes on the server. Thus our ROR provides a reference to a specific object on the server, and each such object provides a different service. For instance, if a client had some other class that implemented, say a PI computing operation, then the server creates an instance of this class and also a corresponding ROR when the server is started. Finally, the server 'binds' all these ROR's to the registry through the registry service.
   Nonetheless, our implementation makes it easy to extend file download in the future for the class files. We can also extend our server to accept user requests with new classes and add them dynamically to the registry.
   We also said that the application server does preliminary processing on user requests. In this case, it plays the role of a dispatcher. It unmarshals client side requests for invoking a method on a server object, along with its parameters. It then creates a 'Servant' which is actually responsible for executing the method. The unmarshalling by the server is on a serializable object sent by the client, which in turn contains all the information about the remote method invocation that the client wants the server to perform.

2. Registry Service and Registry:
   The registry service, instantiated by the application server, runs in a separate thread. It communicates with the actual Registry (which is a SimpleRegistry object) to bind ROR's sent by the application server. By binding an ROR, it basically adds the ROR to a table that the registry maintains. This table is a mapping between the ROR and the name of the interface that it implements, for e.g. 'ZipCodeServer'.
   Apart from the bind method, the registry also has a lookup method. The client can use this method to get an ROR corresponding to a particular object (service) on the server. Additionally, we have also implemented methods like 'rebind' and 'unbind' to assign a new value to a service name, or remove a service reference

(ROR) from the registry, respectively. This gives us a way to dynamically rebind or remove objects in the future.

3. Servant (Thread):
   The dispatcher (application server) unmarshals a client request, and first gets from its own table the actual object that the client is refering. It then creates a new thread (servant) to implement this method. The servant thus takes the object, method name, and arguments, and uses Java's reflection tool to invoke the method on the object. In addition, it obtains the return value from the method, and also catches an exception generated by the method (or during the method invocation), marshals it and sends it back to the client. It then exits, thus removing the thread.
   The motivation behind creating a separate servant thread to serve a user request is that the application server can then go back to listening for other user requests without blocking.

Client Side

On the client side, we have the client which interacts with the user. The client obtains a reference to the remote object in the server through a lookup in the registry. This is done through a special message sent by the client to the registry service, which in turn looks up the ROR from the registry and sends it back to the client. The client then invokes the 'localise' method on the remote object reference.

The localise() method returns a stub that acts as the original implementation object for the client. Since we do not have a stub compiler in our implementation, we assume that the client already has the stub class in its classpath. It has exactly the same methods as defined in the client interface and implemented by the client in its class. Hence, the client thinks that it is invoking methods in its class locally. However, it is actually invoking methods on the stub.

The stub replaces all the client methods with custom implementations. We have designed our stub so that every method implementation is replaced with a generic scheme. We create a new serializable object with fields for method name, arguments and argument classes. In addition, it includes a field for the object key (retrieved from the ROR) which uniquely identifies an object on the server. This object is then 'marshaled' in the stub and sent to the client. This forms our communication handling from the client to the server.

The stub then waits for the server to provide a return value, or an acknowledgement of successful method execution on the server. Once this is obtained, it returns to the client with the return value, if any.

Communication Module

As mentioned above, the client stub creates a special object that is used for marshalling the object key, along with the arguments, argument classes and method name for the method that the client wants to execute. The return value from the server (servant) to the client is sent as a single object.

Also, the lookup performed by the client on the registry is done through a new object (a RegistryMsg object) that sends the message ('lookup') to the registry service along with the service name. The registry service uses the same object to communicate the ROR back to the client.

Finally, the bind, unbind and rebind operations available to the application server are also performed using the RegistryMsg class. The client sends the message 'bind', or potentially 'unbind' and 'rebind', the registry service performs the corresponding operation, and sends back an acknowledgement to the applicaiton server.

Remote Exception

In order to communicate an exception on the server side to the client, we provide the remote exception class. Our implementation provides the client with the ability to review any runtime exception that occurs on the server. The remote exception is communicated to the client by the servant in the same manner as a return value. The client stub checks specifically for a remote exception and throws it to the client. The client then has the ability to deal with it the way it wishes. Please note that at various places, we have printed the entire stack trace of an error, but we have done so with some leading message, and at least by catching the exception first (including in the client that interacts with the user).

Design Decisions

1. **Unimplemented Features**
   a. Stub Compilers are not implemented as it was not required. Instead we have precompiled stubs for the remote object types that wish to provide.
   b. Also, we do not provide a mechanism for downloading class files, we assume the required class files are present on the class-path of both server and client. However, such a mechanism can be easily provided by allowing the server/client to receive .class files through HTTP.
   c. We have also not implemented distributed garbage collection.
   d. Our RMI server does not maintain per-client state and hence does not guarantee safe execution when multiple clients invoke multiple instances of the same remote object/service-type. The rationale behind maintaining a single object instance per service-type is to prevent multiple copies of potentially large resources such as a database on the server.

2. **Untested Features**
   Though we have implemented the 4 methods - bind(), lookup(), rebind() and unbind() for the RMI registry, we do not test the method rebind() and unbind() through our example applications, as we assume static binding - we bind all our remote services/objects when the RMI server initializes.

3. **Single Dispatcher instead of multiple Skeletons**
   Similar to Java 2 onwards, we do not have skeletons on the server for each remote object. The entire functionality of receiving messages, unmarshalling contents, getting local object reference from the extracted remote object reference, method names, parameters and passing it all to the RMI Servant is contained in a single RMI dispatcher. The benefit we get is lesser number of components in the system leading to lower probability of bugs - though this can lead us to lose some concurrency.

4. **Information Hiding for RMI Registry**
   We have a separate RMI Registry service thread, which holds a reference to the actual RMI registry object. This Registry service is responsible for providing the API through the functions lookup, bind, rebind, unbind. This allows us to prevent application clients/servers from directly accessing the registry and hence leads to a secure implementation.

5. **Different message types for communication with RMI server and registry**
   We implement different message types for communication with the RMI server and RMI registry as both are separate entities and communicate different types of information.

6. **No explicit Communication Module**
   We handle all our network communications through the client stubs and dispatcher (server side) instead of a separate class (communication module). The advantages of such a separate module can be seen in a scenario when we are trying to maintain multiple network sessions between multiple client/server pairs - it could be beneficial to have a single module with a single network connection that buffers and routes all communication among these entities.

7. **Flexibility to catch remote exceptions**
   Our RMI implementation provides the ability for throwing/catching remote exceptions through the RemoteException class. The RMI servant catches any exception from the remote method executing on the server, wraps it in a RemoteException object and sends it back to the client stub, which then throws back the remote exception to the application client. Our RMI software requires any remote object (i.e. one that implements the YourRemote interface) to throw a RemoteException. This is checked while initializing the remote object .

8. **Separate package for common classes - rmiservice.rmi.comm**
   We include the classes that are common to both client and server in a different package called ‘rmiservice.rmi.comm’ - however this is just an implementation convenience and has no effects on the system. This package also contains classes (and ADTs) shared between the client and the server. This, again, is for convenience, and we can separate it into the client and server packages with duplicate copies without any issues.

Build/Test Instructions

Build

The build instructions can be found in the README file provided with the src folder.

Test

For testing, we provide two applications that an application program has designed. These include the ZipCodeServer and the CalculatePi API's.

1. The ZipCodeServer's client reads a list of cities and zip codes from a file. It then initializes a list, which in turn is initialized on the server. The client can now find a particular zip code using the name of the city, find all pairs of zip codes and cities, and print all zip codes and cities. Since all the computations (search mainly) take place on the server, the client's resources are spared.
   Using the ZipCodeServer, we are able to demonstrate the main purpose of our RMI facility, coupled with the ability to communicate between the client and the server successfully, and show that we are able to return computational results back to the client.

2. The CalculatePi client asks the user for an integer digit, and calculates the value of Pi up to those many decimal digits. We have taken the code (logic) for generating Pi till *n* decimal digits from the code on the Java RMI page. However, this code is just the Pi calculation logic, and does not play any role in our RMI facility. With this code, we can shift very time consuming computations on the server, and hence prove that an RMI is indeed very useful.
   In addition, we have provided the facility for you to test our Remote Exception generation mechanism. If you provide a negative number of digits for Pi calculation, then the client's implementation on the server throws a Remote Exception, which is conveyed back to the client. The client can also see the stack trace printed in its console.

Instructions for the Application Programmer

In this section, we provide some instructions to an application programmer who wants to run an application using our RMI facility.

1. Create the interface (YourApp) that your application implements, and place it in the rmiservice.rmi.comm directory. Make sure that this interface extends the YourRemote interface in order to be able to bind it to the registry. Also, make sure that all methods throw RemoteException. This is necessary to communicate remote errors back to the client.
2. Create the implementation in a YourApp_Impl class and put it in the rmiservice.rmi.server directory. YourApp_Impl should implement the YourApp interface.
   In YourApp_Impl, write the code that you would have otherwise implemented in the client if RMI was not available.
3. Create a client for YourApp and place it in rmiservice.rmi.client directory. This client communicates with the registry, and obtains the remote object reference (ROR) corresponding to YourApp_Impl from the registry. Obtain the server IP, port, object key from the ROR.
4. Create a YourApp_stub class that implements YourApp, and place it in the rmiservice.rmi.client directory. In YourApp_stub, use the server IP, port and object key obtained from ROR to communicate with the server. In the constructor, create an instance of the ClientRmiMsg class. This will be used send message to the server. Now, in every method, initialize the fields of ClientRmiMsg with appropriate values of the method name to be invokes along with the argument list and classes. Also encapsulate the object key derived from the ROR. You can now send this object to the server, which will unmarshall it, execute the method and pass back the return value to the stub. Hence, wait for the server to return an Object. Make sure to check if this object is null or a RemoteException. If the latter, then throw it back to the client program. If it is the actual return value, then return that to the client program.
5. In addition, for any custom type (ADT) that YourApp_stub, YourApp client and YourApp_Impl all use, put it in the rmiservice.rmi.comm directory.
6. Calling the localise method on the remote object reference will return an instance of YourApp_stub, and you can now use this to make remote method invocations on the server.