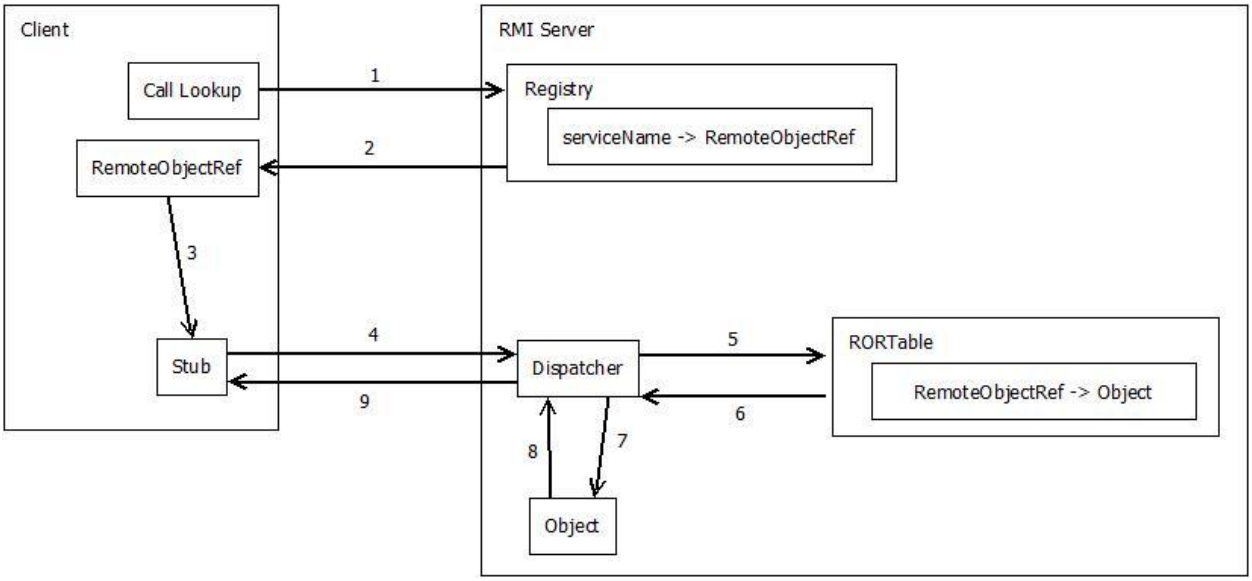
15-440 (Fall 2014)

Lab 2: RMI Facility

Derek Tzeng

Design



My RMI facility allows clients to remotely invoke methods on registered objects on the RMI server.

The server has a registry which contains a mapping of all the service names to their respective RemoteObjectRef's. It supports binding, lookup, rebinding, and unbinding of the services. The server also houses a dispatcher which receives invocation requests from the client stubs and takes care of unmarshalling before and marshalling after the invocation of the desired method.

On the client side, the user is supplied with an interface which allows the lookup of remote services and localization of remote references into stubs which handle all communication with the server.

Now, we go into more detail on the architecture of a RMI workflow. Before the client is able to perform any RMI's, it must first use LocateRegistry's static method getRegistry which connects to the server and finds the port of where the registry is running. After this, a remote invocation follows these steps, as according to the diagram above:

1. Looks up the desired service on the registry using Registry's "lookup()" method.
2. The "lookup" method will return the RemoteObjectRef that corresponds to the service.
3. After getting the RemoteObjectRef, the user can now use "localize" to generate a stub. The user can now call methods on this stub just as if he were calling it on the original

object. The communication to the server is abstracted away from the user. We delve under this abstraction in the next steps to explore the architecture hidden underneath.

4. Suppose that the user calls a method on the stub. The stub will create a new RMIMessage which contains the RemoteObjectRef, the method name, and its parameters. If any of the parameters are remote objects themselves, then they will be replaced by its reference using the "getRef()" method. After this marshalling phase, the RMIMessage will be sent to the dispatcher on the RMI server.
5. The dispatcher receives this RMIMessage and first looks up the actual object reference in the RORTable. If the reference does not exist, then it will create a new implementation and add it into the table.
6. The dispatcher receives the actual object reference.
7. The dispatcher unmarshals the parameters from the RMIMessage and invokes the method on the actual object reference. If any of the parameters are RemoteObjectRef's, then it will make sure to localize them first.
8. The dispatcher receives the return value or exception of the method invocation.
9. If the return value is a remote object, then the dispatcher will make sure to marshal it into a RemoteObjectRef before sending it back to the stub. The stub will unmarshal it back into a remote object once receiving the return object.

I decided to use a dispatcher rather than a skeleton to compliment each stub. This makes my design less repetitive, as I can factor out the common functionality of getting the local object reference and invoking the specified method using Java's reflect package.

Building the Project

In my handin directory at "`/afs/andrew/course/15/440-f14/handin/lab2/dtzeng`", run the following script to build the jar:

```
chmod +x build.sh

./build.sh
```

This will create a jar file called "RMI.jar". You may copy this jar to any desired directory for testing below.

Running the Examples

1. Find 2 GHC machines that are up and running. I will call these 2 machines GHC_SERVER and GHC_CLIENT. Once logged into these 2 machines, "cd" to the directory where the jar was copied above.

2. On GHC_SERVER, start up the RMI server

```
java -cp RMI.jar rmi.server.RMIServer
```

Mark down the second port number displayed, where the server was started. I will call this PORT.

3. On GHC_CLIENT, run the first test

```
java -cp RMI.jar rmi.tests.OpNumsTest GHC_SERVER PORT
```

This test will add a new reference of an OpNum to the registry, retrieve it, and check the results of all the operations.

4. On GHC_CLIENT, run the second test

```
java -cp RMI.jar rmi.tests.ContainerTest GHC_SERVER PORT
```

This test will add a new reference of a Container to the registry, retrieve it, and call the getObject() and setObject() methods. This test makes sure that remote objects retain their state between remote invocations.

5. On GHC_CLIENT, run the third test

```
java -cp RMI.jar rmi.tests.ConcatRefsTest GHC_SERVER PORT
```

This test will add new references to Containers and ConcatRefs to the registry, and retrieves them. This test makes sure that the marshalling and unmarshalling of remote object arguments/returns work correctly.