THE REPORT: RMI

15-440 Project 2

Vansi Vallabhaneni (vvallabh)
Tomer Borenstein (tborenst)

How to Use

Initialization:

1. Create an interface which represents your class and have it extend *MyRemote* interface.
2. On the server, create a new *ServerHandler* object and pass in the port you want it to listen on.
3. Register your classes with *ServerHandler.registerClass()* and pass in the class and the interface it implements.
4. Create and register your remote objects using *ServerHandler.registerObject()* and pass in the object, the interface it represents and its name.
5. On the client, create a new *ClientHandler* object.
6. Connect to your servers with *ClientHandler.connectTo()* and pass in the hostname and port.
7. Register your interfaces with *ClientHandler.registerInterface()* and pass in the interface.

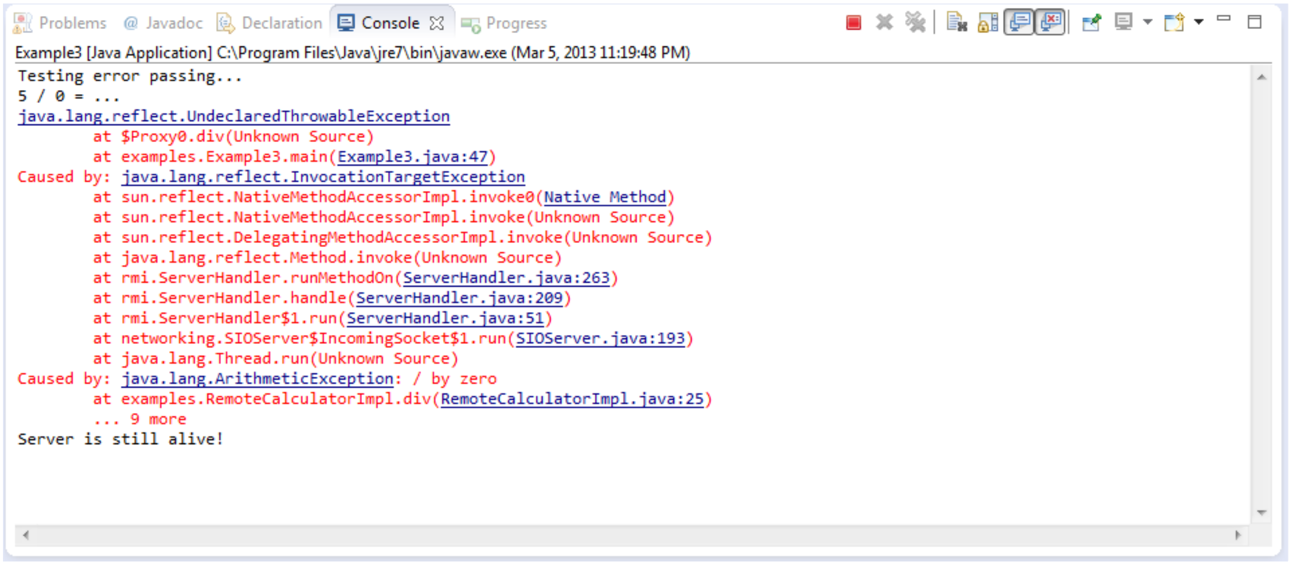
Some Notes:

- In order to retrieve remote services from any server, use the *ClientHandler.lookup()* function by passing it a String of the name of the service. Note that we are not performing a lookup on individual registries, but on all of them. The first service that matches the name will be returned, but since in practice names of services will take the form of “http://serveraddress:port/servicename” this should not matter.

- You can invoke any method that the remote object’s interfaces supports just by treating it as any local object. If the invocation causes an error to be thrown, it will be passed all the way to the client (see below).

- If a remote object returns a reference to another remote object, the client can bind that remote object to a particular name in order to turn it into a service that can be looked up. This is done by calling the *bind()* or *rebind()* function. Our RMI System knows which server to create the service on by looking at the routing information found in the RemoteObjectReference object that is passed into *bind()*. In order to remove a service, use the *unbind()* function.

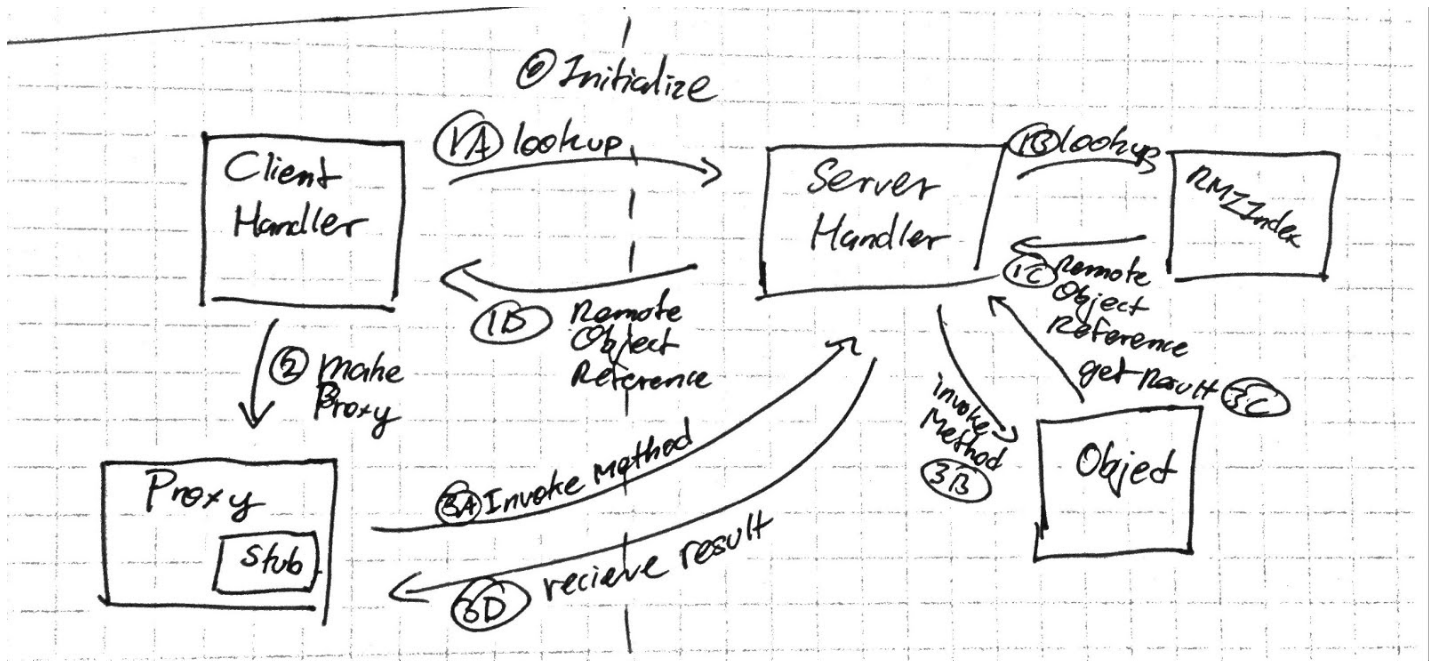
Error Tracing:



```
Example3 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 5, 2013 11:19:48 PM)
Testing error passing...
5 / 0 = ...
java.lang.reflect.UndeclaredThrowableException
	at $Proxy0.div(Unknown Source)
	at examples.Example3.main(Example3.java:47)
Caused by: java.lang.reflect.InvocationTargetException
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
	at java.lang.reflect.Method.invoke(Unknown Source)
	at rmi.ServerHandler.runMethodOn(ServerHandler.java:263)
	at rmi.ServerHandler.handle(ServerHandler.java:209)
	at rmi.ServerHandler$1.run(ServerHandler.java:51)
	at networking.SIOServer$IncomingSocket$1.run(SIOServer.java:193)
	at java.lang.Thread.run(Unknown Source)
Caused by: java.lang.ArithmeticException: / by zero
	at examples.RemoteCalculatorImpl.div(RemoteCalculatorImpl.java:25)
	... 9 more
Server is still alive!
```

Here we try to remotely divide 5 by 0. So this should result in an ArithmeticException on the client, but you will see that we give a lot more information that. This is to make it easier for the application programmer to figure out if the bug is within his code or within our implementation of RMI.

Implementation



Overview of Implementation

Design:

Our goal was to make a very clean and simple api for the application programmer to use, yet keep it easy to maintain.

Client Side:

The ClientHandler is initialized once (even if you have multiple servers) and is the portal through which the application programmer communicates to the registries. This makes it very simple for the application to handle multiple servers. The client side representation of a remote object is the Proxy object. Each Proxy object has a unique stub which packages method invocations into RMIRequest objects, making sure to convert Proxy objects to RemoteObjectReferences. Then the stub sends the requests to the ServerHandler over SocketIO (SIO). The stub then receives RMIResponses over SIO and unpacks the results, making sure to convert RemoteObjectReferences to Proxy objects and throw errors as necessary.

Server Side:

The ServerHandler is the **network facing** interface on the server which handles all client calls. When the ServerHandler receives RMIIndex related class such as (lookup, bind, rebind and unbind) it passes the method calls directly to the RMIIndex. In addition when the ServerHandler receives method invocations from client side stubs, it unpacks the RMIRequest, making sure to convert RemoteObjectReferences to objects by retrieving the references from the RMIIndex and invoking the method with the arguments on the **main object**. Then it packages the response into a RMIResponse, making sure to convert remote objects (objects which implement MyRemote) into RemoteObjectReferences (add them into RMIIndex) and catching any errors, finally ServerHandler packages the results into a RMIResponse and sends the response object back to the client over SIO.

SocketIO (SIO):

In order to make the code more easily maintainable, we've created a little networking package called SocketIO (socket.io), named and designed after the javascript server side environment (node) module that carries the same name. SocketIO allows the programmer to interface with the networking aspects (of any application, really, not just this project) with an event-based programming paradigm in mind, even though Java is inherently threaded. For example:

```
server.on("connection", new SIOCommand(){
      public void run(){
            addSocket(socket); //server logic, say, if you want to keep track of connections
            socket.emit("welcome!", null); //emit takes an event name and an object
      }
});
```

For this project in particular, we also made it possible for the client to send blocking requests to the server. This helped in writing the portion of our RMI system that handles "pass by value" and "pass by reference":

```
Integer result = (Integer) client.request("add", new CalcRequest(2, 3));
```

We like it. It's cool.

Major Design Decisions:

1. Using Proxy and InvocationHandler: We chose to use Proxy and InvocationHandler (stub) to make our implementation dynamic RMI "compilation" so all an application programmer needs to do is initialize the classes and interfaces they implement. This makes it very easy to scale and maintain.
2. Single instance of ClientHandler: rather than have a new instance per server, a single instance makes it easier for the application programmer, and it is not much more work to maintain.
3. Communicate to RMIIndex through ServerHandler: This let us keep all our socket code in one class, making it easier to debug and maintain, also since ServerHandler unpacked and packed messages it made for a better abstraction.
4. SocketIO: Since it is event based and asynchronous, it made the code significantly cleaner and much much easier to maintain and debug.
5. Blocking client side SocketIO: Significantly easier to implement, unfortunately this is a concern and to include some of the more advanced features robust we ran out of time to make this multithreaded.
6. Do not require name to be URL: We think that a long URL is actually harder to remember than any name the user wants to implement.

Future Work

1. Allow servers to communicate and access objects on other servers. This will easily be done by extending our client interface to be a part of the server interface.
2. Implement load balancing on servers (spread around highly referenced objects).
3. HTTP support to download .class files. Since names of well-known services are usually URLs, we could use these in order to download the class files.
4. Decrease the amount of initialization required by the user.
    a. Completely automating the compilation process for stubs. Currently, we require the user to write an interface for his/her remote object, and then we generate the stub based on that interface. Ideally, any class that implements remote should automatically have interfaces generated to create Proxy objects and stubs.
    b. Dynamically add new classes to the client and server from a .class file. This will eliminate the need for clients to register classes. In this case, we would simply automate the registration process on the client when it receives an ROR from the server.
    c. Create a building tool that, with one click, would initialize everything on the server in order for RMI to operate. This will make it even easier for servers to use our system by reducing the amount of work they need to do.
5. Cache responses which are state independent.