RMI Report:

Simon Dai(sdai)

Ken Payson (kpayson)

**General Design**:

We designed our project as a framework that can be easily extended and used by other applications. There are three major components:

**Registry**: The registry Classes are RegistryRequest, RegistryResponse, and RemoteObjectRegistry.

The RegistryRequest and RegistryResponse classes are message wrappers, for requests and responses back from the registry. The request will contain important information, such as the remote object's location (address, port), id, etc; the response has a status (OK) that functions as a confirmation for if the operation succeeds, as well as returning information from a lookup.

The RemoteObjectRegistry is the actual registry server, which continuously accepts connections from clients, who send it a RegistryRequest. It contains information about each remote object in the RMI system. It has a map of each object's unique ID to the location (address, port) of that object. This way, when it receives a lookup request, it can send back the location of the object in question. Also, when making a register or unregister request, it is simply a matter of updating the map.

Finally, we provided a utility class called NetworkUtil, that simply has a bunch of static methods to abstract communication with the registry. If you wanted to register a new remote object, you would simply call NetworkUtil.registryRegister, to lookup call NetworkUtil.registryLookup, etc. It does this by handling all networking messages sent to and from the registry (RegistryRequest, RegistryResponse).

**Remote Objects**: Remote objects are represented by the following: The RemoteObj Interface, the RemoteObjRef class, and the RMIStubInvocationHandler.

The RemoteObj Interface is an interface that every remote object must implement. The only required method of the RemoteObj Interface is getRMIName(), which returns the unique name of the given remote object. When using our framework's dynamic stub generation, this interface should be extended. The subclass interface should contain all of the methods that the specific remote object has. This way, when we return a stub of the specific remote object, the client can use a stub the same way they would a local object. We will expand more on this later, in the dynamic stub generation section.

The RemoteObjRef class is a completely internal way of representing remote objects. It contains the key data associated with a remote object, such as the object's unique name, location, and Interface (for stub generation). It has a localise() method that returns a new stub for the given remote object.

The RMIStubInvocationHandler is essentially our framework's stub abstraction. Instead of compiling stubs for each remote object, we dynamically generate a "InvocationHandler" which functions the exact same way as a stub. This abstraction is what allows us to dynamically generate stubs. This class handles any method calls made to our "stub" by making the appropriate remote method call, and returning the result/throwing an exception as appropriate.

**Remote Method Invocation**: To make remote method calls work, we have a RMIMessage class as a wrapper for messages sent to: RMIHandler, which is the server that handles RMI calls.

The RMIMessage class simply contains information about an RMI call, such as the type of message (INVOKE, RETURN, EXCEPTION), and the necessary information about each type (for example, and INVOKE message should contain the name of the object whose method we are invoking, the name of the method, the parameters to that method, and the types (Classes) of the given parameters).

The RMIHandler class is the proxy server that replaces the remote object's skeleton. It handles all RMI calls to a given node, regardless of which remote object you want on that node. This is accomplished by each RMIHandler instance keeping track of all remote objects on that node, with a map from IDs to local RemoteObj objects. The RMIHandler unmarshalls the message, and converts any RemoteObjectRef parameters to their stub versions, so that they can be used as if they were local objects. It is of note that we spawn threads to handle each connection request, so it is possible that multiple clients want to invoke different methods on the same remote object. This could be problematic for example if you had a remote bank account that tried to make concurrent withdrawals and purchases. So, how our RMIHandler accounts for this is whenever we try to invoke a method on an object o, we synchronize on o (use o, the object itself, as the lock). This allows us to still allow concurrent RMI calls (across connections) as long as the remote objects whose methods are being invoked are different.

A client can obtain a stub by calling the static method getRemoteObject() in RMIHandler, which handles all of the communication with the registry, and localises the remote object, returning the stub. This way the client doesn't need to worry about any network communication with the registry.

**Features/Implementations/Requirements/Problems**:

Each remote object has a unique String ID associated with it (Note that this ID must be generated by the client, see usage section for details). This way, we can invoke remote methods. Importantly, we can use any RemoteObj (remote objects) as parameters or return types interchangeably with local (serializable) objects (note this is handled between our dynamic stubs and RMIHandlers, which manage the marshalling and unmarshalling).

Next, we have a fully functional registry server, which nodes should use to register their own remote objects, and lookup the locations of other remote objects that they want to use.

Our framework also has complete dynamic stub generation, which we will talk about in the next section.

Problems with this design are that it requires the client to generate interfaces for every type of remote object they want to use. Also, we don't generate class files for stubs, but rather generate instances at runtime, so the user does not have a concrete stub for usage outside of our framework, so if the use wants to keep making stubs, we would have to continuously generate them, instead of the user just recycling a previously generated stub class. However, this is still a clean and robust way to ensure correctness in our RMI system.

Note that functions such as equals, hash, and clone are not implemented correctly. This challenge with this is that we would need to ensure that the default behavior works with the RMI address as opposed to the default behavior in both the stubs and actual object. If the user were to implement these functions themselves, that would even further complicate functional consistency with the stubs. Therefore, we decided it was out of the scope of this project.

**Dynamic Stub Generation:**

Our framework does not have a stub compiler; instead, we dynamically generate stubs at runtime. In order for our framework to generate stubs, we require each remote object to create its own interface (we will refer to it as [InterfaceType]) that extends RemoteObj, and contains all methods that the remote object has. This is important so that when we return a stub, the client can use the stub in the same way they would use a local version of that class (Both stubs and local versions are of type [InterfaceType], so the same methods can be used). We don't actually generate an explicit stub class, instead we generate a [InterfaceType] proxy object, which isn't exactly the same, but has the same behavior. This is done using Java's Proxy library, which generates a dynamic object at runtime, and our RMIStubInvocationHandler, which actually defines the behaviors of the proxy "stub." The InvocationHandler does all the marshalling in the remote method call, and sends the marshalled RMIMessage to the RMIHandler at the location of the remote object. Then, when the RMIHandler sends back the result of the method call (return or exception), the InvocationHandler then either returns the appropriate object (could be another stub if the return type was remote), or throws the exception that the remote object raised.

**Using our framework**:

To use our framework, first the user must make some number of remote objects. This is accomplished first by making an interface for each remote object that extends RemoteObj; the interface should contain every method in that remote object. The user can also have some number of serializable objects. Then, once they have some remote objects, to run their code they should first start a registry on some machine, then start RMIHandlers at every node that has a remote object. Whenever a node wants to instantiate a remote object, they must register that object to the registry. Also, when a remote object dies, that object should be unregistered from the registry. Now, any node can get a stub for any remote object by calling RMIHandler.getRemoteObject(), and then use that stub as if it were a local object.

**Libraries/System Requirements:**

We only use jdk1.7. Everything in the rmi\_framework package should be sufficient to run your own example code, similarly to the way we build our examples (see example package).

**Running our Example:**

The first example is run through the Main and Main2 classes. (run Main first, then run Main2, then hit enter in Main to let it know that Main2 is running). This example involves two remote object types, StringReverse, and GlobalStringReverse (again, these are the interfaces describing each remote object's behavior). We have StringReverseImpl and GlobalStringReverseImpl as the local implementations of the given interfaces. Our StringReverseImpl first contains a field appendage which we append to the end of any reversal operation. The reverse function of StringReverse takes in a string to reverse and an Integer value i, which reverses the first i characters of the given string, then appends the appendage. GlobalStringReverseImpl has a integer field that determines the number of characters to be reversed. The globalReverse method takes a list of strings and a StringReverse object, and maps the reverse function onto each element of the list using the integer field as the parameter to reverse.

When the example is run, Main first creates a registry, then creates a RMIHandler (call this node 1). Then, it creates a new StringReverse object (remote) and registers it to that RMIHandler. Then, it waits for Main2 to be started. When Main2 is started, it first gets a stub for the remote StringReverse object on node 1, then calls the reverse function. Then, it starts its own RMIHandler (call this node 2), registering a new GlobalStringReverse object on it. Back to Main, we now create a new list of strings and attempt to call globalReverse on node 2, passing in the list and the local StringReverse object. All of the code up to this point should work as intended. Then, we modify the list so that calling globalReverse will throw an exception, and Main should print the stacktrace of the exception, which we can see is the exception thrown by the remote object.

The final example is run through a class MainStub. This runs the RMI system on a single machine in the same process, but still behaves in a manner that works with multiple machines across a network. A remote interface is defined for a Database, and an implementation TreeDatabase. We create two separate RMIHandlers as well as creating the Registry. One database is registered per RMIHandler. Then, we simply call a variety of functions that invoke both local and stub methods, in addition to more complicated tasks such as passing in RemoteObj's as arguments, returning RemoteObj's, and remotely invocating methods that throw errors. We can see from the output

1 John

2 Alex

3 Kevin

4 Ma-the-vaa

5 Jake

6 Josh

7 Chris

1 John

2 Alex

3 Kevin

4 Ma-the-vaa

5 Jake

6 Josh

7 Chris

Exception properly passed!

that the databases were properly merged in addition to the exception being proxy-thrown by the stub caller.