**Overall Design Overview**

Our RMI Implementation works in the following manner:

*Data Transportation:*

When a client needs to send data to a server, the data is packaged in an *RMIMessage* object. This object is able to store a method name, an array of parameters, and an object key that unique identifies this object. It is also able to store an exception and a return value. When the server needs to respond to a client, it packages the necessary information back into the *RMIMessage* it received and sends it to the client.

*Remote Object References:*

Our *RemoteObjectReference* class stores four vital bits of information while representing a remote object:

1. IP Address – the host where the remote object is located
2. Port – the port on which the server that contains the remote object is listening
3. Object Key – a unique identifier for the remote object
4. Remote Interface Name – the name of the remote interface the remote object implements

The *RemoteObjectReference* class also has a localize function, which allows it to generate a stub using the Remote Interface Name. This stub localizes the remote object and allows for communication with the server that contains the remote object.

*Registry:*

Rather than have an individual registry on each server, our implementation includes a global registry. This registry must be running in order for the RMI to work properly. Our server communications class, RMI440, takes a registry hostname and port in the command line and attempts to connect to the registry. Clients should operate the same way, making their initial lookups in this registry.

The *RegistryImpl* class, which implements the Registry440 interface, uses a *HashMap* to store *RemoteObjectReferences* that can be looked up by a String name. Adding an object to the registry happens with a rebind function, similarly to Java’s RMI Implementation.

A server or client can attempt to locate a registry at a specific host and port through a static function contained within the LocateRegistry class. LocateRegistry returns an instance of RegistryCom440, which also implements the Registry 440 interface. This class is in charge of communicating with the RegistryImpl, and carrying out lookup and rebind requests through these communications. In this way, the client and server need not know what is happening behind the scenes between these two classes. It only needs to know that it has a class that implements Registry440.

*Server:*

All major server-side operations are carried out through the *RMI440* class. There should be one instantiation of *RMI440* for each remote object. This class takes the following command line arguments:

* Initial Class Name – the object for which a client may be accessing
* Registry Host – the host of the global registry
* Registry Port – the port that the global registry is listening on
* Service Name – the name the client will use to identify this object

An instance of RMI440 connects to the registry, creates an instance of the initial class, creates a mapping of a *RemoteObjectReference* to this local object, and binds the service name to the *RemoteObjectReference* in the registry. Once this is completed, it opens a *ServerSocket* that listens for invocation requests.

Upon hearing a request, a socket is open, and the *RMIMessage* is read in using an *ObjectInputStream*. The data is unmarshalled within the *RMI440* class. Java’s Reflection API is used to turn the String method name and array of parameters into a method invocation on the remote object located within this *RMI440*. The *RMI440* then takes the return value (or exception), packages it back into the *RMIMessage*, sends it to the client, and closes the connection.

*Client:*

Clients are expected to establish a connection with the registry, get a *RemoteObjectReference* from this registry, use the localize function with the *RemoteObjectReference*, and call a function through this localized object. The client need not know what is localized; it just should know that it would implement the correct *RemoteInterface.* In this way, the remote nature of the method invocation is abstracted away from the client.

In reality, the method is called on the appropriate stub, which handles the marshalling of the method into an *RMIMessage*, sending this message to the server, waiting for a response, and unmarshalling the return value or exception.

**Major Design Decisions**

*Use of RMIMessage:*

Our implementation uses a uniform way to transfer data between the client and server. Since only one object is to be sent, rather than multiple bits of information, communications becomes much simpler. Also, the ability of the *RMIMessage* to localize a stub allowed for ease in passing remote references between the client and server.

*Global Registry:*

Our implementation uses a global registry (one for the whole network), rather than having a registry located on each server. This is to make it easier on the client. Rather than needing to know the location of the server that contains each remote object it needs, it only needs to know the location of the global registry. Provided it has the correct service names for each object, it can then perform these lookups in the same place. This is both more efficient, and requires a lot less of the client.

*ReigstryCom and RegistryImpl:*

Our design uses two classes that implement the *Registry440* interface in conjunction with each other to perform the necessary registry operations. The *ReigstryImpl* class actually houses the registry, and is run with its own main method on some host and port. When a server or client wants to connect to this registry, it uses a static method within *LocateRegistry* to obtain an implementation of *Registry440*. What implementation it is should be abstracted away from the user, but in reality, it is a *RegistryCom440*. This class handles all communication with *RegistryImpl* when performing lookup and rebind functionality. This allowed us to separate the functionality of the registry, communications and concrete implementation, into two different classes.

*RMI440 containing main method:*

In our implementation, in order to start a server, an RMI440 is instantiated with command line arguments that dictate the remote object the server is wrapping and the name that this object will be known by in the registry. This allows the user to create true remote objects, and not worry about properly creating the server functionality. The user can limit the development of remote objects to exactly the methods that are needed of the object. Remote objects become treated as true objects, rather than servers. We felt like this abstraction was important to understanding the purpose and functionality of the RMI.

*RMI440 unmarshalling method invocations:*

Our method for packaging data, along with our use of Java’s Reflection API, made it unnecessary to use skeletons to unmarshall invocation requests. Our implementation for unmarshalling these requests is purely mechanical, and does not depend on the parameters involved or the remote interface the request was made for. Therefore, for simplicity, we kept all unmarshalling requests, and marshalling return values and exceptions, within our *RMI440* class.