**Project #2: Design and Implementation of a RMI Facility for Java**

1. Clearly explain your design and illustrate its use, being sure to highlight any special features or abilities
   1. **RMIRegistry:** The RMI registry maintains the remote reference information of the services that are present in the distributed. The server (proxy dispatcher) registers the remote reference with the RMI Registry which is looked-up by the SampleClient. The RMI registry will allow the proxy dispatcher to bind an object so that when the SampleClient looks up the remote reference with a “key name” it will provide the information that is needed. The information of the remote object will be packed in an object called RemoteRef. This will contain the class name, interface name, lookup name and the necessary IP address and the port number of the proxy dispatch. With this information the Proxy stub compiler will be able to create the stub object and remotely execute methods. The RMI registry will handle three types of actions that are “bind, rebind, and lookup”. The rebind will occur if there are registered remote references the in RMI registries with the same “key name”, if this occurs the RMI registry will send out an exception so that the Proxy dispatcher could send out the rebind signal. If the look up happens from the Sample Client the RMI registry will send out the RemoteRef back to the Sample Client
   2. **Proxy Dispatcher:** The proxy dispatch will perform two main actions. The first action will be to bind the remote reference objects to the RMI registry. The second action will be to handle the remote reference methods calls from the Simple Client. After the proxy dispatcher executes the method call, it will send back the results to the Simple Client.
   3. **Sample Client:** The sample client will connect to the RMI registry and looking up the remote reference. If there is a bind reference it will use the proxy stub compiler to create the object and perform actions.
   4. **Proxy Stub Compiler (Runtime stub generation):** The stub compiler will generate a client stub at runtime after doing a remote lookup from the RMIRegistry. The Remote Reference object is teased apart and using the Proxy class in java a stub is generated. This client side stub then marshals the information of the method call and sends it over to the client. The created client side stub objects are cached as they are created so this action is not repeated for further calls to the server.
   5. **Signals**
      1. *Ack LookUp Signal:* This signal will be an acknowledgement from the look up signal. This will contain the remote reference information that will be used in the proxy stub compiler.
      2. *Ack Signal:* This signal will be an acknowledgement for the bind, rebind signal.
      3. *Bind Signal:* This signal will contain the remote reference information from the proxy dispatcher.
      4. *Loop Up Signal:* This signal will be a look up request from the Simple client.
      5. *Rebind Signal:* This signal will trigger a rebind on the RMI registry.
      6. *Remote Exception Signal:* This signal will handle the exceptions from the remote locations. If an exception is triggered the remote exception signal will be created with the exception information.
      7. *Invoke Method Signal:* This signal passes information about the method to be executed on the server side and the necessary parameters packaged in an object. This information is unpacked on the server-side and then used to actually call a method.
      8. *Invocation Response Signal:* This signal comprises of the results (on success) or exception when something fails on the server side. The exception is captured in this signal and sent over to the client side or the results is capture packaged in this signal sent over.
   6. **Helper Utility:** This was design to handle the communication and various signals that are used for communication in the system. It will have two methods that will be send and receive. This will be used in all components so that we will have a consistent method to communicate.
2. **Special Features or Abilities**
   1. **Runtime Stub Compiler:** We build a runtime stub compiler instead of generating the stub at compile time. Wherever the client needs to call for a service (call a remote method) we get the RemoteRef from the RMIRegistry and generate the stub to make a remote call. The stubs generated are cached on the client so that they could be re-used again and again for consecutive call.

**Benefits of a dynamic Runtime Stub Compiler:**

* + 1. Due to runtime generation of stub, different variants of the stub could be generated depending on the current needs of the application.
    2. Since stubs are generated at runtime there is no space to for stubs on disk.
    3. Runtime stubs can be portables across platforms or similar modifications can be made.

1. Describe the portions of the design that are correctly implemented, that have bugs, and that remain unimplemented.
   1. The design for the project two, we followed the basic idea that was introduced to use by in the project description with modification. One major this that is a bit different is that we separated the RMI registry from the server. To start the whole system, we need to boot up the RMI registry first. Then start up the proxy dispatcher, as the proxy dispatcher is started it will connect to the RMI registry and bind the necessary remote references. After binding everything, it will start to accept connection from the client so that it could execute the remote invocations. For the client side, it will start up and reach out to the RMI registry looks up the necessary remote reference so that the remote reference information could be used in the stub compiler. We are assuming that the interface file will be present on the client side so that we could initialize the interface and pass it to the stub compiler.
   2. **Not Implemented**
      1. Distributed Garbage collection (on the RMI registry): The RMI registry does not have the functionality to validate the liveness of the remote reference. The remote reference will only be updated with the rebind signal if the proxy dispatcher rebinds with the same “key name” again. There is no service running on RMIRegistry side that does distribute garbage collection.
      2. Automatic retrieval (download) of the interface file for the stub compiler on the client side. (We are keeping the interface is already present on client side)
      3. Maintaining state/session on server-side for clients: All the methods calls on server-side are stateless meaning that the server doesn't remember anything about the client making requests. Server doesn't maintain some session or information about client. Each request is independent.
      4. If there are some connection exceptions while executing the remote methods, we do not have the ability to trigger another re-look up so that we could recreate the proxy object and transparently follow-up the execution.
2. **Tell us how to cleanly build, deploy, and run your project**

**(Please make sure the ports you use are not already in use)**

* 1. Download the source code and ***cd***to the ***Final\_Lab2*** directory (on the command prompt)
  2. Compile the system by running the following command:

**make**

c. To run the RMIRegistry run the following command:   
**java -cp ./RMIRegistry abhi.registry.RMIRegistry 5000**

d. To run the Server run the following command:  
**java -cp ./Server abhi.dispatcherserver.ProxyDispatcher 6000 127.0.0.1 5000**

d. To run the Server run the following command:  
**java -cp ./Client abhi.client.SampleClient 127.0.0.1 5000**

[NOTE: We have not implemented for any client input for the addition or subtraction services we are providing because we wanted to focus more on stub compiler than handling various user inputs for the assignment. We have just hard-coded some values on client side to call add and subtract methods with some parameters]

1. Highlight any dependencies and software or system requirements.
   1. There are no dependencies in the program. The only requirement will be the java 1.70 and the open ports for the connections.
2. Tell us how to run and test your framework with your two examples.
   1. Regular Happy Path
      1. Start up the RMI registry following with a proxy dispatcher.
      2. Execute the simple client and valid the result.

(The Client called on two remote calls. One interface for Addition and one interface for Subtraction)

* 1. Re-bind Testing (Scenario in mind – Server dies and/or another server comes up again)
     1. Start up the RMI registry with one proxy dispatcher and execute the simple client.
     2. Notice that the first proxy dispatcher will react to the remote method call.
     3. Start up another proxy dispatch. Notice that they will be an exception while binding and the second proxy dispatcher will send out the rebind.
     4. Start up a simple client again notice that the second proxy dispatcher will react to the remote method call.