**Appendices M**

**Remote Method Invocation**

1. **Set up Server**

To implement RMI I needed to amend the ServerImpl class created in iteration 1, so that the ServerImpl class registers itself with the RMI registry so that clients are able to locate the server registry when they want to connect.

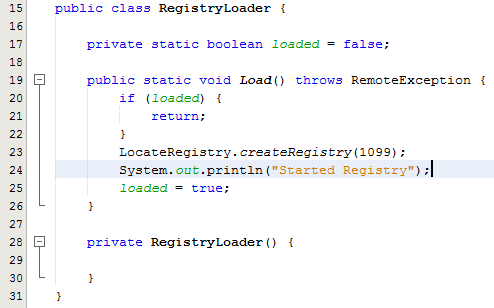


Fig. 1 – RegistryLoader class

As you can see from Fig. 1, I decided to create a RegistryLoader class which deals with creating the registry, and setting it up on the local host to deal with requests on the specified port, to do this I invoked LocateRegistry.createRegistry() and supplied the port number that the server should communicate through. I used the singleton pattern to ensure that if the registry had already been created with the RMI registry then the server class is unable to try and register again.



Fig. 2 – ServerImpl class signature (extending UnicastRemoteObject and implementing Server)

I then amended the ServerImpl class to extend UnicastRemoteObject as shown in Fig. 2, which enables me to then export a Server instance of ServerImpl, which can then be registered with the RMI registry, and allow clients to retrieve the server stub to then invoke remote methods.

Due to design decisions which will be discussed later in this section, I also had to amend each class within the system model that was going to be made available to the client, to make them extend UnicastRemoteObject, however only the server stub needs to be registered with the RMI registry as explained earlier, the server will act as the controller, between the client package “View” and the rest of the server package “Model”.

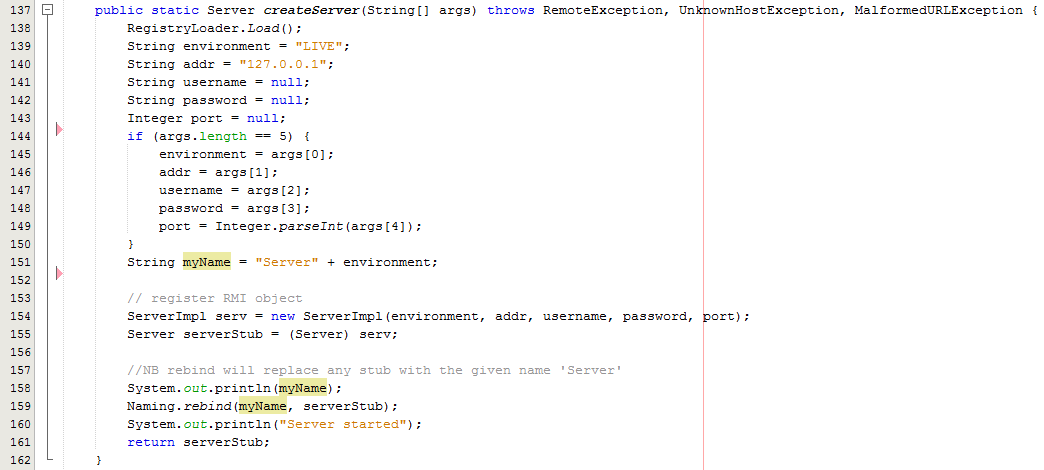


Fig. 3 – Extract from ServerImpl – createServer()

I then amended the ServerImpl.createServer() method to invoke RegistryLoader.Load() method as shown in Fig. 3, which will invoke the RegistryLoader class, I then extract the information supplied from the client from the String array called args supplied as a parameter, if the String array has 5 elements then each element is supplied to the variables declared and some are then passed as parameters to invoke the ServerImpl constructor and create a new ServerImpl instance.

I then invoke the Naming.rebind() method which deals with binding the specified name of my server to a new server stub, which is a Server (Remote) instance of the ServerImpl instance just created. The name of the server is Server + ‘environment’, (where environment is either, LIVE, TRAIN or TEST), which enables this server software to be run on 3 different hosts and act as a live, train or test environment for ‘MSc Properties’.



Fig. 4 – createNote() method signature (throws RemoteException)

I also had to make any remote methods, or any methods dependent on a remote method for the server side classes, throw a Remote Exception to the client invoking the method, as shown in Fig. 4.

1. **Set up Client**

Once the Server side coding was complete I then needed to create a client package, and as explained before, I already had a common package which consisted of any classes or interfaces which would be common between both the server side and client side packages, this meant that I firstly needed to add the common package into the newly created client package.

The first step I took was to create a ClientImpl class within the client package, and an interface for ClientImpl which I added to the common package, ClientImpl would then implement Client. I then needed to get the server stub that was registered with the RMI registry, which will be the object that the client invokes methods on to interact with the system.

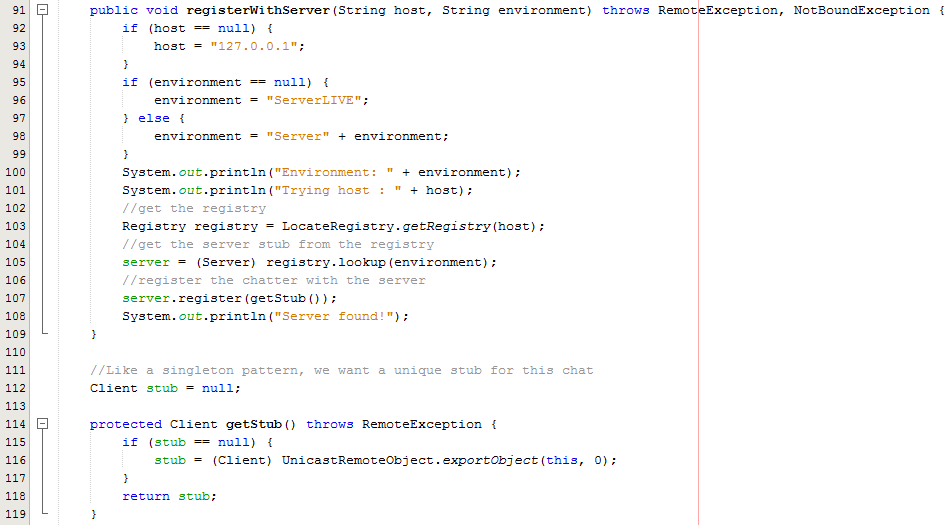


Fig. 5 – Extract from ClientImpl – registerWithServer() and getStub()

As shown in Fig. 5, to allow the Client to have a stub of the Server, I needed to get the Registry using the IP address of the Server by invoking the static method LocateRegistry.getRegistry(), and then invoke the lookup() method on the Registry object, passing the Server name as a parameter, which returns an instance of the Server, I then invoke register() on the returned server object, and pass a Client stub as a parameter, this Client stub will also be stored at the Server.

Once the Client has a stub of the server, the Client is then able to invoke any methods that is available through the Server interface.

1. **Push vs Pull**

As I decided to use the pull data exchange concept to implement the networking functionality RMI, as explained, it required me to make all of the classes that are going to be passed to the client extend UnicastRemoteObject and implement the Interface within the common package for the given class. I then had to make all of the interfaces within the common package extend Remote, enabling the interface to act as a Remote object casing for the UnicastRemoteObject, as shown in Fig. 6.



Fig. 6 - AccountInterface interface signature (extending Remote)

Additionally, as shown in Fig. 6, all of the methods defined in the Interface will have to throw a Remote Exception, enabling the client to invoke a method on the remote object.

Now the set-up is configured to enable the client to invoke any methods that are defined in the interfaces within the common package, on remote objects, I am going to explain how the networking functionality ensures the Server implementation is controlling any updates to the system through the pull data exchange concept.

For me to implement the system in this way, I only define getter methods within any of the interfaces defined within the common functions package, with the exception of the Client and Server interfaces. This ensures that all updating functions are defined within the Sever interface, meaning if a user wants to perform any update functions on the system, they have to invoke a method on the Server object, and then the Server object will either perform the function depending on if the input is valid, and return a positive response, or return a negative result if the update should to the system should not be performed.

As you can see from Fig. 7, I have defined an updatePerson() method within the Server interface which allows a user of the system, on the client side to update a person object in which they may currently have an instance of, as a remote object, and because there is no update methods defined within the PersonInterface interface, clients are not able to invoke the updatePerson method within the Person class, shown in Fig. 8, which means that the client package is forced to invoke a method on the Server as shown in Fig. 9, which means that the ServerImpl class can then control the updates and ensure that the updates are valid and that the Database can then be updated as shown in Fig. 10.



Fig. 7 – Extract from Server Interface – updatePerson()



Fig. 8 – Extract from Person class – updatePerson()

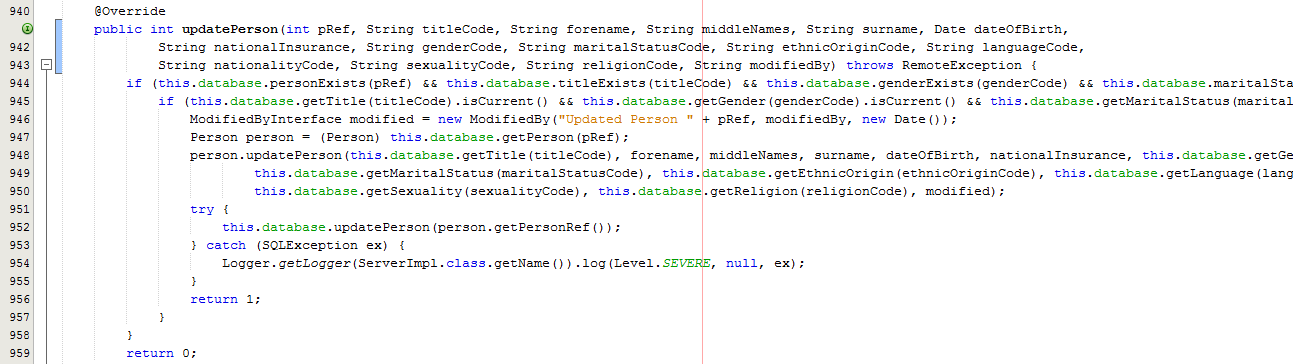


Fig. 9 – Extract from ServerImpl class – updatePerson()

As you can see from Person.updatePerson() in Fig. 8, the method does not throw a RemoteException, and does not override a method within PersonInterface interface, meaning that the client package is not able to invoke the method on a Remote person object, and as explained, forces the client package to invoke the ServerImpl.updatePerson() method through the Server interface, as shown in Fig. 9, thus ensuring the ServerImpl has control over any updates, and ensures that any updates invoked on the server are valid and can then update the MySQL database defined in iteration cycle 2, as you can see from the Database.updatePerson() method being invoked after the ServerImpl object updates the person by invoking Person.updatePerson() on the person object.