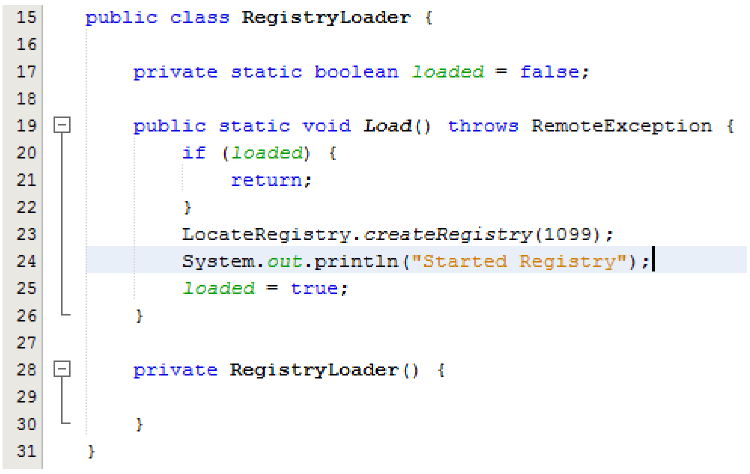
Appendices L

Remote Method Invocation

1. **Set up Server**

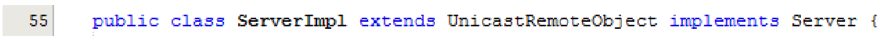
To implement RMI I needed to amend the ServerImpl class created in iteration 1, so that the ServerImpl class registers itself with the RMI registry so that clients are able to locate the server when they want to connect.



```
public class RegistryLoader {

    private static boolean loaded = false;

    public static void Load() throws RemoteException {
        if (loaded) {
            return;
        }
        LocateRegistry.createRegistry(1099);
        System.out.println("Started Registry");
        loaded = true;
    }

    private RegistryLoader() {

    }
}
```

Fig x – RegistryLoader class

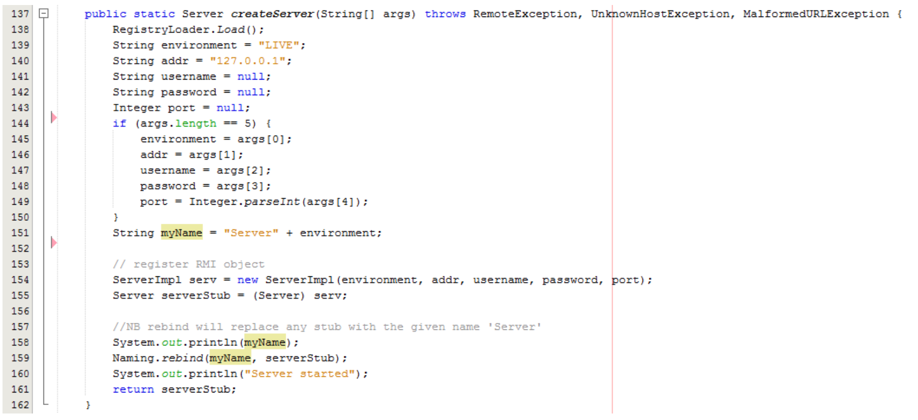
As you can see from Fig x, I decided to create a RegistryLoader class which deals with creating the registry, and setting it up on the local host to deal with requests on the specified port, to do this I invoked LocateRegistry.createRegistry() and supplied the port number that the server should communicate 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.



```
public class ServerImpl extends UnicastRemoteObject implements Server {
```

I then amended the ServerImpl class to extend UnicastRemoteObject, 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".

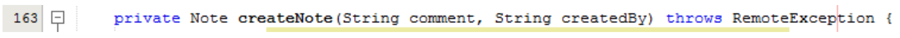


```
    public static Server createServer(String[] args) throws RemoteException, UnknownHostException, MalformedURLException {
        RegistryLoader.Load();
        String environment = "LIVE";
        String addr = "127.0.0.1";
        String username = null;
        String password = null;
        Integer port = null;
        if (args.length == 5) {
            environment = args[0];
            addr = args[1];
            username = args[2];
            password = args[3];
            port = Integer.parseInt(args[4]);
        }
        String myName = "Server" + environment;

        // register RMI object
        ServerImpl serv = new ServerImpl(environment, addr, username, password, port);
        Server serverStub = (Server) serv;

        //NB rebind will replace any stub with the given name 'Server'
        System.out.println(myName);
        Naming.rebind(myName, serverStub);
        System.out.println("Server started");
        return serverStub;
    }
```

Fig x – Extract from ServerImpl – createServer()

I then amended the ServerImpl.createServer() method to invoke RegistryLoader.Load() method, 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'.



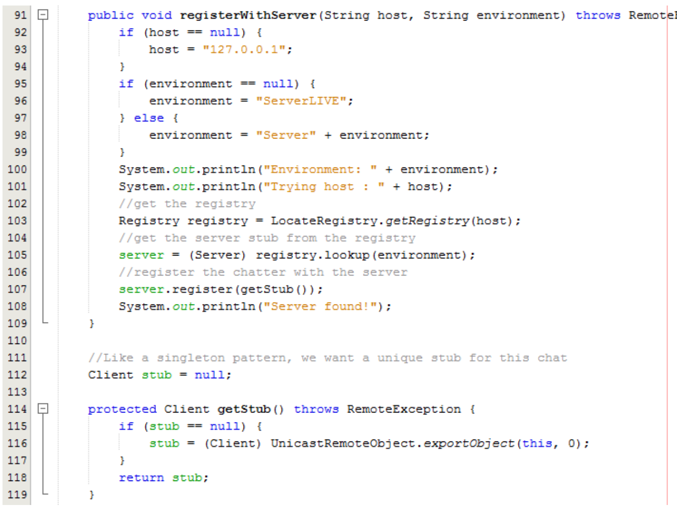
```
    private Note createNote(String comment, String createdBy) throws RemoteException {
```

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

2. 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.



```java
    public void registerWithServer(String host, String environment) throws RemoteException, NotBoundException {
        if (host == null) {
            host = "127.0.0.1";
        }
        if (environment == null) {
            environment = "ServerLIVE";
        } else {
            environment = "Server" + environment;
        }
        System.out.println("Environment: " + environment);
        System.out.println("Trying host : " + host);
        //get the registry
        Registry registry = LocateRegistry.getRegistry(host);
        //get the server stub from the registry
        server = (Server) registry.lookup(environment);
        //register the chatter with the server
        server.register(getStub());
        System.out.println("Server found!");
    }

    //Like a singleton pattern, we want a unique stub for this chat
    Client stub = null;

    protected Client getStub() throws RemoteException {
        if (stub == null) {
            stub = (Client) UnicastRemoteObject.exportObject(this, 0);
        }
        return stub;
    }
```

Fig x – Extract from ClientImpl – registerWithServer() and getStub()

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.

3. Push vs Pull

As I decided to use the Pull method to implement