



## INDIVIDUAL ASSIGNMENT

CT024-3-DCOMS

DISTRIBUTED COMPUTER SYSTEMS

NP3F2304IT

HAND OUT DATE:  AUGUST 2023

HAND IN DATE:  NOVEMBER 2023

WEIGHTAGE:  50%

---

### INSTRUCTIONS TO CANDIDATES:

- 1 Assignment is to be submitted through online submission (Moodle).
- 2 Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing).
- 3 Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld.
- 4 Cases of plagiarism will be penalized.
- 5 Where the assignment should be submitted online (Moodle), the link to the source code and other relevant sources (where appropriate) should be attached together with the submission.

## **Acknowledgment**

I gratefully acknowledge the help and assistance for the completion of our assignment to **Asia Pacific University (APU), Lord Buddha Education Foundation (LBEF) Campus**, and our DCOMS Module leader Mr. Shambhu Gautam for providing us with valuable guidance and constructive feedback at every stage of this project. His expertise and support have been invaluable in the successful completion of this project.

This report's content addresses all the criteria for the assignment, and we put forth a lot of work to finish this task. As a result, this report follows a well-structured flow of content based on project requirements.

Sujan Shrestha

Bsc.IT 5th Semester

## Table of Contents

<b>1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	Background and Context of RMI application development project.....	1
1.2	RMI Framework and its Role in Distributed Systems.....	2
1.3	Cloud Computing and Virtualization .....	2
<b>2</b>	<b>Objective and Goals.....</b>	<b>4</b>
<b>3</b>	<b>System design.....</b>	<b>4</b>
3.1	Overview of RMI application architecture and components .....	4
3.2	Interaction of RMI Application Components .....	6
<b>4</b>	<b>Implementation.....</b>	<b>8</b>
4.1	RMI Application Development Process with Code Snippets.....	8
4.1	Use Case Diagram.....	26
<b>5</b>	<b>Testing.....</b>	<b>27</b>
5.1	Integration Testing.....	27
<b>6</b>	<b>Deployment in the Cloud: .....</b>	<b>27</b>
6.1	Benefits and Challenges of Cloud-Based Deployment: .....	28
6.2	Future Enhancements: .....	28
<b>7</b>	<b>Conclusion .....</b>	<b>28</b>
<b>8</b>	<b>References.....</b>	<b>29</b>

## Table of Figures

Figure 1: Interaction of RMI Application Components .....	6
Figure 2: Data Flow in RMI.....	7
Figure 3: Defining Remote Interface I.....	8
Figure 4: Defining Remote Interface II.....	9
Figure 5: Extending Implementation class with Interface.....	9
Figure 6: RMI Method Implementation class I .....	10
Figure 7: RMI Method Implementation II .....	10
Figure 8: RMI Method Implementation III.....	10
Figure 9: RMI Method Implementation IV.....	11
Figure 10: RMI Method Implementation V .....	11
Figure 11: RMI Method Implementation VI.....	12
Figure 12: RMI Method Implementation VII .....	12
Figure 13: RMI Method Implementation VIII .....	13
Figure 14: RMI Method Implementation IX.....	13
Figure 15: RMI Method Implementation X.....	14
Figure 16: RMI Method Implementation XI.....	14
Figure 17: RMI Method Implementation XII .....	14
Figure 18: Login UI .....	15
Figure 19: Sign Up UI .....	15
Figure 20: Forgot Password UI.....	16
Figure 21: User Details UI.....	16
Figure 22: Admin User Management UI I .....	17
Figure 23: Admin User Management UI II.....	17
Figure 24: Client-Side Implementation I .....	18
Figure 25: Client-Side Implementation I .....	18
Figure 26: Client-Side Implementation II.....	19
Figure 27: Client-Side Implementation III.....	19
Figure 28: Client-Side Implementation IV .....	20
Figure 29: Client-Side Implementation V.....	21
Figure 30: Client-Side Implementation VI .....	22
Figure 31: RMI Registry.....	22
Figure 32: Server Setup .....	23
<b>Figure 33: Client Setup.....</b>	<b>25</b>
Figure 34: Use Case Diagram.....	26
Figure 35: Integration Testing.....	27

# **1 Introduction**

In the world of Java programming, RMI stands for Remote Method Invocation which is referred as an important application programming interface which allows one computer or JVM which stands from Java Virtual Machine to connect with objects on another JVM via a distributed network. RMI serves as a conduit for creating distributed Java applications through simplicity of method calls. RMI is also considered to be matched effectively with the concept of a Java Remote Procedure Call (RPC) but extends its capabilities by allowing the delivery of one or more objects alongside request. This complex system mainly uses two basic components to facilitate smooth remote interactions between application: the 'stub' and the 'skeleton' (Awati, 2019).

## **1.1 Background and Context of RMI application development project**

The RMI application development project began in broader concept of distributed computing, an approach that is used when expanding program capabilities by distributing components over multiple machines. Java perfectly aligns with RMI, making it a popular alternative for designing distributed systems as the synergy between java and RMI, multiple components of the application can communicate continuously, with servers and clients engaging coherently. RMI takes advantage of Java objects' ability to communicate openly over a network, which contributes to the project's robust and versatile architecture (MAASSEN, 2001).

RMI can be used for a variety of purposes such as developing client-server applications, the building of distributed databases, and the setting up of remote administration and control systems. The RMI technique is based on an object-oriented paradigm, which allows for seamless functionality in a distributed context. This object-oriented paradigm, which forms the foundation of Remote Machine Interface (RMI), allows objects dispersed across a network to access their methods. This not only simplifies the design and conceptualization of distributed systems, but it also improves the application's overall efficiency and coherence.

The security of any distributed systems is crucial and RMI addresses such concerns by executing comprehensive security standards. Specific use cases must be defined within the context of the RMI project to tailor the development process to the CKF Group Malaysia's particular goals and objectives. Given the project's distributed nature, maintaining safe interactions between components is critical. To protect RMI interactions and the overall system's integrity, the project will need to investigate and implement best practices.

## **1.2 RMI Framework and its Role in Distributed Systems**

Remote Method Invocation (RMI) is a key Java technology for the construction of distributed systems, allowing for seamless communication between Java objects. This technology allows objects to interact with one another and invoke methods remotely, even if they are on different Virtual Machines or environments. The essence of RMI is its ability to remotely activate methods or objects, allowing dispersed systems to run more efficiently. RMI becomes the backbone of programs that demand collaboration between diverse components spread across different computer systems by facilitating communication among objects across different machines (Sharp, 2006).

RMI's critical role in distributed systems can be summarized as follows:

- **Method Invocation and Object Communication:** RMI acts as a conduit for objects to communicate with one another and invoke methods on remote machines. This functionality is important in instances when application components run on separate machines.
- **Bridge between Components and Servers:** RMI acts as a bridge to connect components on the server to different machines. This link enables the invocation of methods, allowing for the cohesive and integrated operation of distributed components.
- **Handling Network Communication Details:** Handling RMI abstracts and managing sophisticated network communication elements, facilitating the design and deployment of distributed systems. This layer hides developers from the complexity of low-level network connection, resulting in a simpler and more efficient development process.
- **Rapid create and Deployment:** RMI's features make it easier to create and deploy distributed systems. Its architecture allows developers to concentrate on the logic and functionality of the program while RMI handles the underlying communication issues, resulting in a more efficient and rapid development process.

## **1.3 Cloud Computing and Virtualization**

Virtualization is a transformational technology allowing numerous virtual machines (VMs) to process on a single physical device by creating various virtual instances of computing resources. The hypervisor, also known as the Virtual Machine Monitor (VMM), regulates resource allocation, and each VM operates independently with its own operating system. It allows effective utilization of resources through pooling, isolating VMs for increased security, and supports backup and scalability capabilities such as snapshots and cloning (Gopalakrishnan, 2015).

To abstract and efficiently utilize real infrastructure, virtual instances of computer resources like servers or storage are created. Its significance in distributed systems includes the following:

- Virtualization enables numerous virtual machines (VMs) to run on a single physical server. This consolidation improves overall performance and energy efficiency by optimizing resource consumption in distributed systems.
- Each virtual instance functions independently, allowing applications to be isolated from one another. This improves distributed system security by avoiding potential breaches in one component of the system from affecting others.
- Virtualization permits the dynamic allocation of resources to different components of a distributed system. This adaptability guarantees that applications obtain the resources they require in real-time, allowing them to adjust to changing workloads.
- Virtualization enables the construction of isolated testing environments, allowing developers to test distributed programs in controlled environments. This shortens the development time and ensures the robustness of distributed systems.

Cloud computing represents a paradigm shift towards the delivery and consumption of computing services as it provides on-demand services via internet connectivity to a shared pool of adjusting computing resources. Users can access services required, offering a cost-effective service. Cloud computing provides widespread network access, allowing for ubiquitous connectivity from a variety of devices. Because of its resource pooling and quick flexibility, it allows for dynamic scaling to suit shifting workloads. Cloud services are classified as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), providing customers and companies with a variety of possibilities (Elsenpeter, 2010).

Cloud computing provides a scalable and adaptable architecture that enables distributed systems to respond to changing workloads. Its primary significance is found in:

- **Scalability:** Cloud computing allows for the smooth scaling of resources in response to demand. This ensures that applications in a distributed system can handle higher workloads without losing performance.
- **Resource Efficiency:** Cloud computing allows resources to be allocated and used on-demand. As a result, resource efficiency improves because distributed systems can dynamically allocate and deallocate resources as needed, optimizing utilization.

- **Fault Tolerance:** Redundancy and data backup technologies are frequently used by cloud providers. This improves distributed system fault tolerance, maintaining data integrity and system availability even in the face of hardware failures or disruptions.
- **Global Accessibility:** Cloud services enable worldwide access to distributed systems, allowing users to access apps and data from any location with an internet connection. This matches the scattered nature of modern apps.

## **1.4 Objective and Goals**

- Create a user-friendly and efficient Product Delivery System (PDS) with RMI and Java.
- Enable functionalities like user account registration, authentication, profile management, purchase management with necessary information.
- Create an effective order processing system that provides order confirmation, status updates, and thorough reports to customers.
- Use RMI technology for effective communication among distributed components.
- Quality requirements, such as usability, maintainability, and heterogeneity, should be used to evaluate the implemented application.

# **2 System design**

## **2.1 Overview of RMI application architecture and components**

In the RMI architecture, numerous distributed components contribute to the development of a distributed system that allows seamless communication between clients and servers. These components are important to the overall functionality of the Remote Method Invocation process.

Remote Interface:

The remote interface refers to a base element in the RMI architecture that determines the structure for remote methods that are required to be invoked across different Java Virtual Machines (JVMs). By extending the 'java.rmi.Remote' interface, it establishes a standardized blueprint for method signatures, ensuring consistency among the distributed components (Graba, 2007).

Remote Objects:

At the core of the RMI architecture are remote objects, Java entities which executes the specified remote interface. These objects are hosted on servers and play a vital role in implementing the



actual logic of remote methods. When a client initiates an RMI call, it is these remote objects that handle execution of methods and provide the corresponding response.

#### Stubs and Skeletons:

Stubs and skeletons act as intermediaries facilitating the communication between clients and servers. On the client side, stubs serve as local proxies to the remote objects, serializing method calls and directing them to the server. On the server side, skeletons receive these calls, invoke the corresponding methods on the actual remote objects, and manage the serialization of results back to the client (Vecchio, 2002).

#### RMI Registry:

The RMI Registry refers to a central directory registry, providing a mechanism for servers to register their names or URLs. Clients can then validate specific remote objects using these identifiers, obtaining references that are essential for creating method calls. This centralized registry streamlines the process of object identification and retrieval.

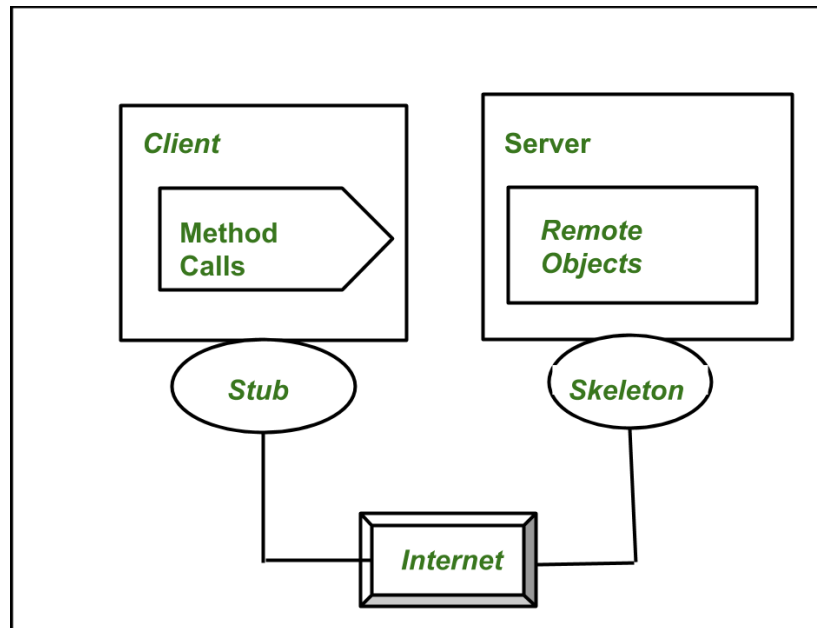
#### RMI API:

The RMI API, residing in the 'java.rmi' package, is a comprehensive set of interfaces and classes that equip developers with the tools needed to implement and manage RMI applications. It encompasses functionalities related to remote object management, naming, and registry interactions, forming the backbone of RMI application development.

#### Java Virtual Machine:

The Java Virtual Machine (JVM) is a crucial component responsible for executing Java code on both the client and server sides. It ensures the proper functioning of remote objects and the execution of RMI method calls in a distributed environment. The JVM creates and manages the runtime environment for RMI applications, playing a fundamental role in their seamless operation.

## 2.2 Interaction of RMI Application Components



*Figure 1: Interaction of RMI Application Components*

### i) Stub:

The stub acts as the gateway on the client side, symbolizing the remote object and facilitating outbound requests. When a method is invoked on the stub, a sequence of actions is passed. It initiates a connection to the remote Virtual Machine (JVM), marshals the parameters by writing them to the remote JVM, waits for the response, decodes the return value upon receipt, and ultimately displays the result to the user. The stub plays a critical role in managing the communication flow from the client to the server, ensuring proper data transmission and reception.

### ii) Skeleton:

On the server side, the skeleton plays a role of a receiver of incoming requests sent towards the server-side object. Once an inbound request is received, the skeleton performs critical tasks. It extracts the arguments of the remote method, invokes the actual method on the true remote object, captures the result, and marshals it back to the caller. The skeleton serves as the intermediary on the server side, orchestrating the execution of remote methods and handling the communication flow between the server and the client.

The interaction between the stub and skeleton is a vital aspect of the RMI architecture. The stub on the client side marshals parameters and sends requests to the skeleton on the server side. The skeleton, in turn, extracts arguments, invokes the actual method on the remote object, captures the

result, and marshals it back to the stub. This interaction ensures the seamless execution of remote methods and the exchange of data between the client and server components.

### iii) Transport Layer:

The transport layer plays a pivotal role in controlling the references called by clients for remote objects within the RMI architecture. It handles the transmission of data between components, ensuring the integrity and efficiency of communication.

### iv) Remote Reference Layer (RRL):

In charge of establishing and maintaining connections between the client and server, the Remote Reference Layer (RRL) is an important component in the RMI architecture which oversees all existing and currently active connections, ensuring a robust and reliable linkage between the client and server for seamless remote method invocations (Tejera, 2005).

## Data Flow and Communication Protocols in RMI

- **Client Initiation:** A client initiates a request for a remote call on a specific object.
- **Serialization of Method Parameters:** The method parameters and names are serialized, converting them into a format suitable for transmission.
- **Client Stub Handling:** The client stub manages the serialization of parameters, preparing them for network transmission.
- **Network Transmission:** The serialized data is transmitted through the network to the server, where the remote object resides.
- **Server Processing and Response:** The server processes the received data and generates a response.
- **Client Receives Response:** The client receives the response from the server.
- **Deserialization by the Client:** The client deserializes the received result.
- **Stub Delivers Result to Client Code:** The client stub delivers the deserialized result to the client code.
- **Successful RMI Invocation:** The RMI invocation is successfully completed, and the data is now available locally for further use.

Figure 2: Data Flow in RMI

The above figure depicts the data flow process visualization, which visualized the flowing phases of the communication between the client and server components in a Java RMI application environment.

### JRMP (Java Remote Method Protocol):

The Java Remote Method Protocol (JRMP) is used in the context Java RMI for secure transmission of data between Java Virtual Machines. JRMP is a combination of rules that control the serialized

transmission of data between clients and servers. This communication protocol enables the smooth and continuous invocation of methods on remote objects, making sure that a continuous network connection within the RMI framework. Java RMI achieves a standardized and secure communication route by conforming to JRMP, which is required for the reliable operation of distributed applications.

### 3 Implementation

In the online shopping application development, the following are the implementations done from my side in our group project. The user authentication and entire User management including profile management, login, signup was part of components done by me where I have executed the principles of an RMI environment as explained below:

#### 4.1 RMI Application Development Process with Code Snippets

An RMI application system is developed by the following process:

i) Definition of Remote Interface

The first thing done was to define the remote interfaces required for communication between the client and the server in the RMI application.

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Interface.java to edit this template
 */
package dao;
import java.rmi.Remote;
import java.rmi.RemoteException;
import javax.swing.JTable;
/**
 *
 * @author aryan
 */
public interface UserDao extends Remote{
    public boolean isUserEmailExist(String email) throws RemoteException;
    public boolean isUserPhoneExist(String phone) throws RemoteException;
    public void insertUser(int id, String username, String email, String pass, String phone,
        String seq, String ans, String address1, String address2) throws RemoteException;
    public int getMaxUserRow() throws RemoteException;
    public int getUserId(String email) throws RemoteException;
    public String[] getUserValue(int id) throws RemoteException;
    public void deleteUser(int id) throws RemoteException;
    public void updateUser(int id, String username, String email, String pass, String phone,
        String seq, String ans, String address1, String address2) throws RemoteException;
    public void getUsersValue(JTable table, String search) throws RemoteException;

    // public void getProductsDValue(JTable table, String search) throws RemoteException;
}
```

*Figure 3: Defining Remote Interface I*

```

/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Interface.java to edit this template
 */
package dao;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 *
 * @author Administrator
 */
public interface LoginInterface extends Remote {
    public boolean loginUser(String email, String password) throws RemoteException;
    public boolean loginSupplier(String email, String password) throws RemoteException;
    public boolean loginAdmin(String email, String password) throws RemoteException;
}

```

*Figure 4: Defining Remote Interface II*

Here, the above two figures shows interfaces created fir Login and User management component of the application.

ii) Implementation of remote methods

After the definition of remote interfaces, next this to be done is implementing the remote interfaces to create remote object. This Java interface includes method declarations for remote invocation by clients. The contents for the methods declared in the remote interface is developed in this part of the code.

```

public class UserDaoImp extends UnicastRemoteObject implements UserDao{
    public UserDaoImp() throws RemoteException {
        super();
    }
}

public class LoginDaoImp extends UnicastRemoteObject implements LoginInterface{
    public LoginDaoImp() throws RemoteException {
        super();
    }
}

```

*Figure 5: Extending Implementation class with Interface.*

```

//get user table max row
@Override
public int getMaxUserRow() throws RemoteException{
    int row = 0;
    try {
        st = con.createStatement();
        rs = st.executeQuery(string:"select max(uid) from user");

        while (rs.next()) {
            row = rs.getInt(1);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
    return row + 1;
}

```

*Figure 6: RMI Method Implementation class I*

```

//check email already exists
@Override
public boolean isUserEmailExist(String email) throws RemoteException {
    try {
        ps = con.prepareStatement(string:"select * from user where uemail = ?");
        ps.setString(1, string:email);
        rs = ps.executeQuery();
        if(rs.next()){
            return true;
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
    return false;
}

```

*Figure 7: RMI Method Implementation II*

```

@Override
public boolean isUserPhoneExist(String phone) throws RemoteException {
    try {
        ps = con.prepareStatement(string:"select * from user where uphone = ?");
        ps.setString(1, string:phone);
        rs = ps.executeQuery();
        if(rs.next()){
            return true;
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
    return false;
}

//insert data into user table database

```

*Figure 8: RMI Method Implementation III*

```

//insert data into user table database
@Override
public void insertUser(int id, String username, String email, String pass, String phone,
    String seq, String ans, String address1, String address2) throws RemoteException
{
    String sql = "insert into user values (?, ?, ?, ?, ?, ?, ?, ?, ?)";
    try {
        ps = con.prepareStatement(string:sql);
        ps.setInt(i: 1, i1: id);
        ps.setString(i: 2, string:username);
        ps.setString(i: 3, string:email);
        ps.setString(i: 4, string:pass);
        ps.setString(i: 5, string:phone);
        ps.setString(i: 6, string:seq);
        ps.setString(i: 7, string:ans);
        ps.setString(i: 8, string:address1);
        ps.setString(i: 9, string:address2);
        if(ps.executeUpdate() > 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "User added successfully");
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
    }
}

```

*Figure 9: RMI Method Implementation IV*

```

//update user data
@Override
public void updateUser(int id, String username, String email, String pass, String phone,
    String seq, String ans, String address1, String address2) throws RemoteException {
    String sql = "update user set uname=?, uemail=?, upassword=?, uphone=?, usecqus=?, uans=?, uaddress1=?, uaddress2=?";
    try {
        ps = con.prepareStatement(string:sql);
        ps.setString(i: 1, string:username);
        ps.setString(i: 2, string:email);
        ps.setString(i: 3, string:pass);
        ps.setString(i: 4, string:phone);
        ps.setString(i: 5, string:seq);
        ps.setString(i: 6, string:ans);
        ps.setString(i: 7, string:address1);
        ps.setString(i: 8, string:address2);
        ps.setInt(i: 9, i1: id);
        if (ps.executeUpdate() > 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "User data successfully updated!");
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
    }
}

```

*Figure 10: RMI Method Implementation V*

```

//delete user
@Override
public void deleteUser(int id) throws RemoteException{
    int x = JOptionPane.showConfirmDialog(parentComponent: null, message: "Are you sure to delete this account", title: "Delete Account");
    if(x == JOptionPane.OK_OPTION){
        try {
            ps = con.prepareStatement(string:"delete from user where uid=?");
            ps.setInt(i: 1, i1: id);
            if(ps.executeUpdate()>0){
                JOptionPane.showMessageDialog(parentComponent: null, message: "Account deleted!");
            }
        } catch (SQLException ex) {
            Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
        }
    }
}
}

```

*Figure 11: RMI Method Implementation VI*

```

//get user id
@Override
public int getUserId(String email) throws RemoteException {
    int id = 0;
    try {
        ps = con.prepareStatement(string:"select uid from user where uemail = ?");
        ps.setString(i: 1, string:email);
        rs = ps.executeQuery();
        if(rs.next()){
            id = rs.getInt(i: 1);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
    return id;
}

```

*Figure 12: RMI Method Implementation VII*



```

//get user data
@Override
public void getUsersValue(JTable table, String search) throws RemoteException{
    String sql = "select * from user where concat(uid, uname, uemail) like ? order by uid desc";
    try {
        ps = con.prepareStatement(string:sql);
        ps.setString(i: 1, "%" + search + "%");
        rs = ps.executeQuery();
        DefaultTableModel model = (DefaultTableModel) table.getModel();
        Object[] row;
        while(rs.next()){
            row = new Object[9];
            row[0] = rs.getInt(i: 1);
            row[1] = rs.getString(i: 2);
            row[2] = rs.getString(i: 3);
            row[3] = rs.getString(i: 4);
            row[4] = rs.getString(i: 5);
            row[5] = rs.getString(i: 6);
            row[6] = rs.getString(i: 7);
            row[7] = rs.getString(i: 8);
            row[8] = rs.getString(i: 9);
            model.addRow(rowData: row);
        }

    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
}

```

*Figure 13: RMI Method Implementation VIII*

```

@Override
public String[] getUserValue(int id) throws RemoteException {
    String[] value = new String[9];
    try {
        ps = con.prepareStatement(string:"select * from user where uid = ?");
        ps.setInt(i: 1, i1: id);
        rs = ps.executeQuery();
        if(rs.next()){
            value[0] = rs.getString(i: 1);
            value[1] = rs.getString(i: 2);
            value[2] = rs.getString(i: 3);
            value[3] = rs.getString(i: 4);
            value[4] = rs.getString(i: 5);
            value[5] = rs.getString(i: 6);
            value[6] = rs.getString(i: 7);
            value[7] = rs.getString(i: 8);
            value[8] = rs.getString(i: 9);
        }
    } catch (SQLException ex) {
        Logger.getLogger(name: UserDaoImp.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
    }
    return value;
}

```

*Figure 14: RMI Method Implementation IX*

```

@Override
public boolean loginUser(String email, String password) throws RemoteException {
    try {
        Connection con = MyConnection.getConnection();
        PreparedStatement ps = con.prepareStatement(string: "select * from user where uemail =? and upassword =?");
        ps.setString(i: 1, string: email);
        ps.setString(i: 2, string: password);
        ResultSet rs = ps.executeQuery();
        return rs.next();
    } catch (SQLException ex) {
        ex.printStackTrace();
        return false;
    }
}

```

*Figure 15: RMI Method Implementation X*

```

@Override
public boolean loginSupplier(String email, String password) throws RemoteException {
    try {
        Connection con = MyConnection.getConnection();
        PreparedStatement ps = con.prepareStatement(string: "select * from supplier where semail =? and spassword =?");
        ps.setString(i: 1, string: email);
        ps.setString(i: 2, string: password);
        ResultSet rs = ps.executeQuery();
        return rs.next();
    } catch (SQLException ex) {
        ex.printStackTrace();
        return false;
    }
}

```

*Figure 16: RMI Method Implementation XI*

```

@Override
public boolean loginAdmin(String email, String password) throws RemoteException {
    try {
        Connection con = MyConnection.getConnection();
        PreparedStatement ps = con.prepareStatement(string: "select * from admin where email =? and password =?");
        ps.setString(i: 1, string: email);
        ps.setString(i: 2, string: password);
        ResultSet rs = ps.executeQuery();
        return rs.next();
    } catch (SQLException ex) {
        ex.printStackTrace();
        return false;
    }
}

```


*Figure 17: RMI Method Implementation XII*

These are various methods required for my part of user management. These methods are called via RMI through UI part of the client program.

### iii) Developing the UI for my parts

Here are some snippets of the UI of my components from the RMI application. The figures below include UI of components created with Swing like Login interface, signup interface, forget password interface, user management and profile management interface.

# Welcome



Login As:

☒ User ☐ Supplier ☐ Admin

Email

Password

[Forgot Password?](#)

Login

Don't have an account? [Sign up](#)

Figure 18: Login UI

# SIGNUP



User ID

Username

Email

Password

Phone

Security Question

Answer

Address line 1

Address Line 2

Save

Back

Figure 19: Sign Up UI

# FORGOT PASSWORD

Email

Security Question

Answer

New Password

Save

Back



Figure 20: Forgot Password UI

# USERACCOUNT

User ID

Username

Email

Password


Phone

Security Question

Answer

Address line 1

Address Line 2



Update Profile

Figure 21: User Profile Update UI

MANAGE USERS

User ID

Username

Email

Password

Phone

Security Question

Answer

Address line 1

Address Line 2

Update

Delete

Clear

Search

User ID	Username	Email	Password	Phone	Security Q...	Answer	Address Li...	Address Li...
2	cranky	cranks@g...	sujan	987867878...	What color...	white	thali	sankhu
1	sujan	sujan@gm...	sujan	1234567890	a	a	What is th...	1

Figure 22: Admin User Management UI I

MANAGE USERS

User ID

2

Username

cranky

Email

cranks@gmail.com

Password

\*\*\*\*\*

Phone

98786787887

Security Question

thali

Answer

sankhu

Address line 1

What color is sky?

Address Line 2

white

Update

Delete

Clear

Search

2

User ID	Username	Email	Password	Phone	Security Q...	Answer	Address Li...	Address Li...
2	cranky	s@gmail.com	sujan	987867878...	What color...	white	thali	sankhu

Figure 23: Admin User Management UI II

iv) Developing the client program

A client-side program is created once the remote object and interface have been established. This software interfaces the client with the remote object, allowing users to create and interact with it.

```
private void init(){
    try {
        jTextField1.setBackground(bg: notEdit);
        jTextField1.setText(t: String.valueOf(i: client.getRemoteService().getMaxUserRow()));
        setLocation(x: 450, y: 180);
    } catch (RemoteException ex) {
        Logger.getLogger(name: SignUp.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
    }
}
```

Figure 24: Client-Side Implementation I

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
    if(isEmpty()){
        try {
            int id = Integer.parseInt(s: jTextField1.getText());
            String username = jTextField2.getText();
            String email = jTextField3.getText();
            String pass = String.valueOf(data: jPasswordField1.getPassword());
            String phone = jTextField4.getText();
            String seq = jComboBox1.getSelectedItem().toString();
            String ans = jTextField5.getText();
            String address1 = jTextField6.getText();
            String address2 = jTextField7.getText();

            if(!client.getRemoteService().isUserEmailExist(email)){
                if(!client.getRemoteService().isUserPhoneExist(phone)){
                    extra.insertUser(id, username, email, pass, phone, seq, ans, address1, address2);
                    new Login().setVisible(b: true);
                    this.dispose();
                }else{
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Phone Number already exists!", title: "Warning");
                }
            }else{
                JOptionPane.showMessageDialog(parentComponent: this, message: "Email address already exists!", title: "Warning",);
            }
        } catch (RemoteException ex) {
            Logger.getLogger(name: SignUp.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
        }
    }
}
```

Figure 25: Client-Side Implementation I

```

private void init() {
    try {
        setLocation(x: 450, y: 180);
        uId = client.getRemoteService().getUserId(email: UserDashboard.userEmail.getText());
        value = client.getRemoteService().getUserValue(id: uId);
        setValue();
    } catch (RemoteException ex) {
        Logger.getLogger(name: UserAccount.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
    }
}

private void jTextField9KeyReleased(java.awt.event.KeyEvent evt) {
    try {
        jTable1.setModel(new DefaultTableModel(data: null, new Object[]{"User ID", "Username", "Email",
            "Password", "Phone", "Security Question", "Answer", "Address Line 1", "Address Line 2"}));
        client.getRemoteService().getUsersValue(table: jTable1, search: jTextField9.getText()); // TODOC
    } catch (RemoteException ex) {
        Logger.getLogger(name: ManageUsers.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
    }
}

```

Figure 26: Client-Side Implementation II

```

private boolean check() {
    String newEmail = jTextField3.getText();
    String newPhone = jTextField4.getText();
    String oldEmail = model.getValueAt(row: rowIndex, column: 2).toString();
    String oldPhone = model.getValueAt(row: rowIndex, column: 4).toString();
    if (newEmail.equals(anObject: oldEmail) && newPhone.equals(anObject: oldPhone)) {
        return false;
    } else {
        if (!newEmail.equals(anObject: oldEmail)) {
            try {
                boolean x = client.getRemoteService().isUserEmailExist(email: newEmail);
                if (x) {
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Email already exists", title: "Warning", messageType: JOptionPane.WARNING_MESSAGE);
                }
                return x;
            } catch (RemoteException ex) {
                Logger.getLogger(name: ManageUsers.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
            }
        }
        if (!newPhone.equals(anObject: oldPhone)) {
            try {
                boolean x = client.getRemoteService().isUserPhoneExist(phone: newPhone);
                if (x) {
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Phone number already exists", title: "Warning", messageType: JOptionPane.WARNING_MESSAGE);
                }
                return x;
            } catch (RemoteException ex) {
                Logger.getLogger(name: ManageUsers.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
            }
        }
    }
    return false;
}

```

Figure 27: Client-Side Implementation III

```

private void btnUpdateActionPerformed(java.awt.event.ActionEvent evt) {
    if (isEmpty()) {
        if (!check()) {
            int id = Integer.parseInt(s: jTextField1.getText());
            String username = jTextField2.getText();
            String email = jTextField3.getText();
            String pass = String.valueOf(data: jPasswordField1.getPassword());
            String phone = jTextField4.getText();
            String seq = jTextField8.getText();
            String ans = jTextField5.getText();
            String address1 = jTextField6.getText();
            String address2 = jTextField7.getText();
            try {
                client.getRemoteService().updateUser(id, username, email, pass, phone, seq, ans, address1, address2);
            } catch (RemoteException ex) {
                Logger.getLogger(name: ManageUsers.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
            }
            jTable1.setModel(new DefaultTableModel(data: null, new Object[]{"User ID", "Username", "Email",
                "Password", "Phone", "Security Question", "Answer", "Address Line 1", "Address Line 2"}));
            try {
                client.getRemoteService().getUsersValue(table: jTable1, search:"");
            } catch (RemoteException ex) {
                Logger.getLogger(name: ManageUsers.class.getName()).log(level: Level.SEVERE, msg:null, thrown:ex);
            }
            clear();
        }
    }
}

```

*Figure 28: Client-Side Implementation IV*



```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    if (isEmpty()) {
        try {
            String email = jTextField1.getText();
            String password = String.valueOf(data: jPasswordField1.getPassword());

            LoginDaoImp loginDao = new LoginDaoImp();

            if (jRadioButton1.isSelected()) {
                boolean loggedIn = client.getRemote2Service().loginUser(email, password);
                if (loggedIn) {
                    UserDashboard ud = new UserDashboard();
                    ud.setVisible(b: true);
                    ud.pack();
                    UserDashboard.userEmail.setText(text: email);
                    statistics.user(rs.getInt(1));
                    this.dispose();
                } else {
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Incorrect email or password", title: "Login Failed", messageType: 2);
                }
            } else if (jRadioButton2.isSelected()) {
                boolean loggedIn = client.getRemote2Service().loginSupplier(email, password);
                if (loggedIn) {
                    SupplierDashboard sd = new SupplierDashboard();
                    sd.setVisible(b: true);
                    sd.pack();
                    SupplierDashboard.supplierEmail.setText(text: email);
                    statistics.supplier(rs.getString(2));
                    this.dispose();
                } else {
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Incorrect email or password", title: "Login Failed", messageType: 2);
                }
            }
        } catch (RemoteException ex) {
            Logger.getLogger(name: Login.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
        }
    }
}

```

Figure 29: Client-Side Implementation V

```

private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
    if (isEmpty()) {
        try {
            int id = Integer.parseInt(jTextField1.getText());
            String username = jTextField2.getText();
            String email = jTextField3.getText();
            String pass = String.valueOf(jPasswordField1.getPassword());
            String phone = jTextField4.getText();
            String seq = jComboBox1.getSelectedItem().toString();
            String ans = jTextField5.getText();
            String address1 = jTextField6.getText();
            String address2 = jTextField7.getText();

            if (!client.getRemoteService().isUserEmailExist(email)) {
                if (!client.getRemoteService().isUserPhoneExist(phone)) {
                    extra.insertUser(id, username, email, pass, phone, seq, ans, address1, address2);
                    new Login().setVisible(b: true);
                    this.dispose();
                } else {
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Phone Number already exists!", title: "Warning", messageType: 2);
                }
            } else {
                JOptionPane.showMessageDialog(parentComponent: this, message: "Email address already exists!", title: "Warning", messageType: 2);
            }
        } catch (RemoteException ex) {
            Logger.getLogger(name: SignUp.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
        }
    }
}

```

*Figure 30: Client-Side Implementation VI*

- v) Compiling the source files and client stubs and server skeletons generation:  
The next step is to compile Java source files to build skeletons and stubs for both the remote object class and the client software. This is accomplished by utilizing the javac or Java compiler, which produces class files.
- vi) Starting the RMI Registry: The RMI registry was created to allow users to look for remote objects by name. It's in the same folder as the classes.

```

public static void main(String args[]) {
    try {
        UserDaoImp imp = new UserDaoImp();
        LoginDaoImp lg = new LoginDaoImp();
        PurchaseDaoImp pi = new PurchaseDaoImp();
        SupplierDaoImp si = new SupplierDaoImp();
        ProductDaoImp pri = new ProductDaoImp();
        CategoryDaoImp ci = new CategoryDaoImp();
        Registry reg = LocateRegistry.createRegistry(port: 1099);
        reg.bind(string: "UserDao", remote: imp);
        reg.bind(string: "LoginInterface", remote: lg);
        reg.bind(string: "PurchaseInterface", remote: pi);
        reg.bind(string: "SupplierInterface", remote: si);
        reg.bind(string: "ProductInterface", remote: pri);
        reg.bind(string: "CategoryInterface", remote: ci);

        System.out.println(x: "Server has been successfully initiated..");
    }
}

```

*Figure 31: RMI Registry*

- vii) Starting the remote object and running the client:

The server is launched after the registry has been started, allowing it to hold server information. The client software is run with the necessary configurations. Implementing the RMI registry to look for remote objects and enable method invocation is required.

```
import dao.LoginDaoImp;
import dao.SupplierDaoImp;
import dao.UserDaoImp;
import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class OSServer {

    public static void main(String args[]) {
        try {
            UserDaoImp imp = new UserDaoImp();
            LoginDaoImp lg = new LoginDaoImp();
            PurchaseDaoImp pi = new PurchaseDaoImp();
            SupplierDaoImp si = new SupplierDaoImp();
            ProductDaoImp pri = new ProductDaoImp();
            CategoryDaoImp ci = new CategoryDaoImp();
            Registry reg = LocateRegistry.createRegistry(port: 1099);
            reg.bind(string: "UserDao", remote: imp);
            reg.bind(string: "LoginInterface", remote: lg);
            reg.bind(string: "PurchaseInterface", remote: pi);
            reg.bind(string: "SupplierInterface", remote: si);
            reg.bind(string: "ProductInterface", remote: pri);
            reg.bind(string: "CategoryInterface", remote: ci);

            System.out.println(x: "Server has been successfully initiated..");

        } catch (AlreadyBoundException | RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

*Figure 32: Server Setup*

```

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class OSClient {
    private String host = "127.0.0.1";
    private String serviceName = "UserDao";
    private String service2Name = "LoginInterface";
    private String service3Name = "PurchaseInterface";
    private String service4Name = "SupplierInterface";
    private String service5Name = "ProductInterface";
    private String service6Name = "CategoryInterface";
    private int port = 1099;

    // return UserDao object
    public UserDao getRemoteService() {
        try{

            Registry reg = LocateRegistry.getRegistry(host, port);
            return (UserDao) reg.lookup(string: serviceName);

        } catch (RemoteException | NotBoundException e){
            e.printStackTrace();
            return null;
        }
    }

    public LoginInterface getRemote2Service() {
        try{

            Registry reg = LocateRegistry.getRegistry(host, port);
            return (LoginInterface) reg.lookup(string: service2Name);

        } catch (RemoteException | NotBoundException e){
            e.printStackTrace();
            return null;
        }
    }

    public PurchaseInterface getRemote3Service() {
        try{

            Registry reg = LocateRegistry.getRegistry(host, port);
            return (PurchaseInterface) reg.lookup(string: service3Name);

        } catch (RemoteException | NotBoundException e){
            e.printStackTrace();
            return null;
        }
    }

    public SupplierInterface getRemote4Service() {
        try{

            Registry reg = LocateRegistry.getRegistry(host, port);
            return (SupplierInterface) reg.lookup(string: service4Name);

        } catch (RemoteException | NotBoundException e){
            e.printStackTrace();
            return null;
        }
    }
}

```

```

public ProductInterface getRemote5Service(){
    try{

        Registry reg = LocateRegistry.getRegistry(host, port);
        return (ProductInterface) reg.lookup(string: service5Name);

    } catch (RemoteException | NotBoundException e){
        e.printStackTrace();
        return null;
    }
}

public CategoryInterface getRemote6Service(){
    try{

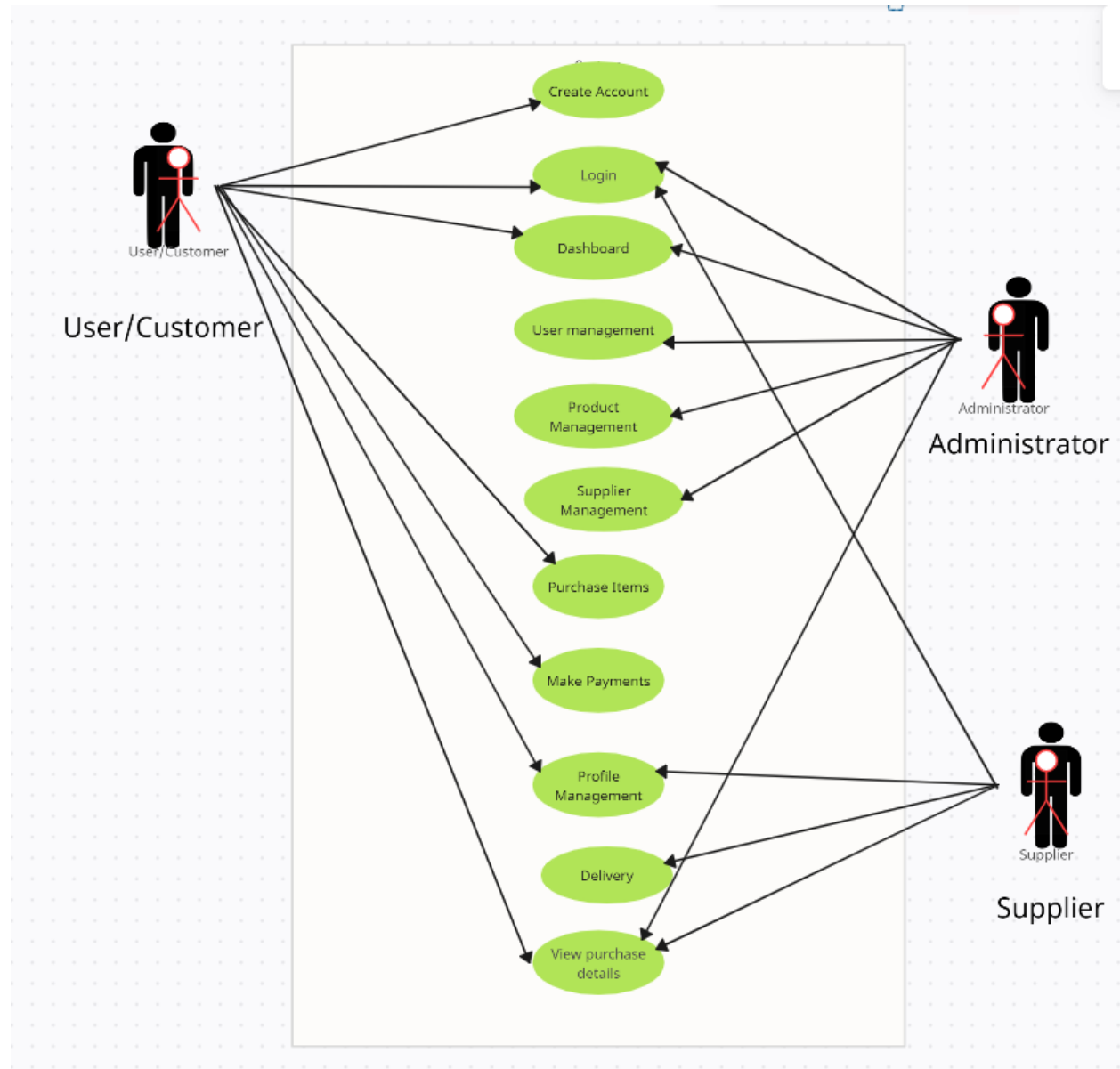
        Registry reg = LocateRegistry.getRegistry(host, port);
        return (CategoryInterface) reg.lookup(string: service6Name);

    } catch (RemoteException | NotBoundException e){
        e.printStackTrace();
        return null;
    }
}

```

***Figure 33: Client Setup***

### 3.1 Use Case Diagram



*Figure 34: Use Case Diagram*

The use case diagram in figure shows the interactions between a user, a supplier, and an administrator in the context of a Online Product Delivery System. The system allows users to create accounts, login, view product details, purchase items, make payments, manage their profiles, and view purchase details. The administrator can manage users, products, suppliers, and deliveries. The use case diagram provides a high-level overview of the system and how it is used by the different actors. It can be used to communicate the system's requirements to stakeholders and to guide the development of the system.

## 4 Testing

In order to ensure the development of this application was done properly, certain testing functions were carried out by my team. Since I have implemented the purchasing or the ordering part of the application, I have performed certain tests regarding this part of application. Here, proper brief regarding the testing of my component is given below:

### 4.1 Integration Testing

As there were different components for each member in our team, I have chosen integration testing as a method of testing my component as it will ensure the specific area of the application is good to integrate with the main application (A study of integration testing and software regression at the integration level, 2002). This testing provided me with brief results of tests performed in the user management and login/signup area of the application. In the following table brief results of the test Is provided.

S.N.	Function	Description
1	Admin/Customer/Supplier Login	This allows the Customer/supplier/admin to login to the system and navigate them to dashboard
2	Customer Signup	This functionality allows customers to register themselves into the system
3	Self Profile Mangement	This allows the user to view and edit their profile details
4	Forgot Password	This allows users to reset their password in case they forget it
5	Admin User Management	This allows admin to view total users and update/delete users in system
6	Admin Supplier Management	This allows admin to view total suppliers and update/delete users in system
7	User Search Functionality	This allows admin to search for any specific customer or suppliers in the system

Expected Result	Actual Result	Status	Priority
All users can be able to login with email and password	All users can login sucessfully	Closed	High
Customers should be able to sign up after filling all the required info	Registration function successfull with required information	Closed	High
Update and delete details of self	Successfull profile Update and Deletion	Closed	Medium
Able to reset the password with correct answer to security question	Able to reset the password with correct answer to security question	Closed	Medium
Admin Should be able to view total users and update/delete users in system	Admin can view total users and update/delete users in system	Closed	High
Admin Should be able to view total suppliers and update/delete users in system	Admin can view total suppliers and update/delete users in system	Closed	High
Admin should be able to search and select specific supplier or customer for edit	Admin can search and select specific supplier or customer for edit	Closed	High

*Figure 35: Integration Testing*

This testing ensures that users can effectively view their profile details and also admins can view accurate user and supplier details, search and add suppliers to the system and search for users' profile along with functionality of adding and deleting users.

## 5 Deployment in the Cloud:

Deployment of the RMI application in environment of cloud computing depends deciding on a cloud service provider, such as AWS or Azure, and then also a suitable architecture plays vital role in this. The utilization of the cloud enhances worldwide accessibility, scalability, and flexibility

along with the Configuration of networks, implementation of load balancing, and ensuring security are all valuable stages. Potential network latency and security concerns are among the challenges that may arise while deploying such RMI applications over the cloud. Overall, cloud deployment improves the overall resource utilization and enables dynamic scaling in response to application demand (Buyya, 2013).

### **5.1 Benefits and Challenges of Cloud-Based Deployment:**

Various benefits such as Scalability, flexibility, cost effectiveness, and worldwide accessibility come with the deployment of such RMI application over the cloud. Organizations can efficiently lower the cost by utilization of a pay-as-you-go model. However, certain difficulties can occur during this implementation which may include network latency and security concerns, particularly when storing and sending data. There can be potential issue of vendor lock-in, underlining the significance of implementing cloud-agnostic methods. Regulation compliance is critical, and careful consideration of these elements assures effective cloud adoption.

### **5.2 Future Enhancements:**

Furthermore, exploring developing cloud computing and virtualization technologies can create the pathway for future enhancements. Integrating enhancements such as serverless computing, virtualization improvements, and support for AI and IoT are all promising areas of investigation which seems difficult but can be developed in the sense to improve the currently existing services in cloud. Furthermore, the application could benefit from blockchain for increased security. By listening to these trends, the RMI application may remain inventive and flexible to changing technology landscapes.

## **6 Conclusion**

The RMI application development project successfully applies over CKF's Product Delivery System issues. Utilization of RMI and deployment of the application over the cloud enhances the efficiency and scalability of the application system. The system's reliability is furthermore confirmed by using different testing methodologies, and future developments are in line with new technology. The project displays the groundwork for developments that can be enhanced in future through cloud computing and virtualization, finally achieving its goals and objectives of enhancing the Product Delivery System.



## 7 References

- A study of integration testing and software regression at the integration level. (2002, August).  
doi:<https://doi.org/10.1109/ICSM.1990.131377>
- Ahuhed, S. (2020). Performance evaluation of Java RMI: a distributed object architecture for Internet based applications. *IEEE Xplore Journal of Computer Science*.
- Awati, R. (2019). Remote Method Invocation (RMI). *Front-end, back-end and middle-tier frameworks*. Retrieved November 2, 2023, from [https://www.theserverside.com/definition/Remote-Method-Invocation-RMI#:~:text=Remote%20Method%20Invocation%20\(RMI\)%20is,JVM%20in%20a%20distributed%20network](https://www.theserverside.com/definition/Remote-Method-Invocation-RMI#:~:text=Remote%20Method%20Invocation%20(RMI)%20is,JVM%20in%20a%20distributed%20network).
- Borner, D. (2021). More efficient serialization and RMI for Java. *Wiley Online Journal for Computing*.
- Buyya, R. (2013). Mastering Cloud Computing. Retrieved November 19, 2023, from <https://www.sciencedirect.com/topics/computer-science/remote-method-invocation>
- Elsenpeter, R. (2010). *Cloud Computing: A Practical Approach*. Retrieved November 19, 2023, from <https://ds.amu.edu.et/xmlui/bitstream/handle/123456789/9207/CloudComputing.pdf?sequence=1&isAllowed=y>
- Gopalakrishnan, V. (2015, February). Network function virtualization. *Challenges and opportunities for innovations*. doi:<https://doi.org/10.1109/MCOM.2015.7045396>
- Graba, J. (2007). *An Introduction to Network Programming with Java*. London: Springer.  
doi:[https://doi.org/10.1007/978-1-84628-630-8\\_5](https://doi.org/10.1007/978-1-84628-630-8_5)
- Hagimont, D., & Boyer, F. (2021). A configurable RMI mechanism for sharing distributed Java objects. *Computer Science Journal for Advanced Computing*.
- MAASSEN, J. (2001). Efficient Java RMI for Parallel Programming. Retrieved November 19, 2023, from [https://scholar.google.com/scholar?hl=en&as\\_sdt=0%2C5&q=RMI&btnG=](https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q=RMI&btnG=)
- Sharp, M. (2006, September). Static Analysis of Object References in RMI-Based Java Software. 664 - 681. doi:<https://doi.org/10.1109/TSE.2006.93>
- Tejera, D. (2005, June 6). Two alternative RMI models for real-time distributed applications.  
doi:<https://doi.org/10.1109/ISORC.2005.62>
- Vecchio, A. (2002, August 6). Embedding remote object mobility in Java RMI.  
doi:<https://doi.org/10.1109/FTDCS.2001.969628>