

|  |  |
| --- | --- |
| **ASSIGNMENT** | |
| **Module Code :** | CSN504 |
| **Module Name :** | Distributed ad Cloud Computing |
| **Course :** | M. Tech. in Computer Science and Networking |
| **Department :** | Computer Science and Engineering |
| **Faculty :** | Faculty of Engineering and Technology |

#### 

|  |  |
| --- | --- |
| **Name of the Student :** | Jhansi V Setty |
| **Reg. No :** | 17ETCS036003 |
| **Batch :** | FET 2017 |
| **Module Leader :** | Mr. Kishore S. M. |

**Ramaiah University of Applied Sciences University House, Gnanagangothri Campus, New BEL Road, M S R Nagar, Bangalore, Karnataka, INDIA - 560 054**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Declaration Sheet | | | | | | | | |
| Student Name | Jhansi V Setty | | | | | | | |
| Reg. No | 17ETCS036003 | | | | | | | |
| Course | M. Tech. in Computer Science and Networking | | | | | Batch | Full-Time / Part-Time 2017 | |
| Module Code | CSN504 | | | | | | | |
| Module Title | Distributed and Cloud Computing | | | | | | | |
| Module Date | 4th June 2018 | | to | | 7th July 2018 | | | |
| Module Leader | Mr. Kishore S. M. | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the student | |  | | | | | Date | 7th July 2018 |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Module Leader and date | | | | Signature of Reviewer and date | | | | |
|  | | | |  | | | | |

# Abstract

This assignment aims at strengthening the concepts of distributed computing by comparing the old architecture with the new distributed architecture. It also aims at providing a basic idea about one of the simplest distributed computing techniques like client server communication through the implementation of a online banking application.

The Chapter 1 focuses on the concept of traditional IT architecture and the latest cloud based architecture. These architectures are compared by listing out their merits and demerits. Later a debate is carried out on whether the cloud can completely replace the traditional architecture. Based on consideration and challenges of the traditional to cloud transition, it is concluded that though cloud has more advantages, it will not replace the traditional IT infrastructure in any near future.

The Chapter 2 contains the design for developing a simple online banking application. Remote Method Invocation is used to implement the application. The components required for the implementation are defined and sequence diagrams and flow charts are drawn to analyse the interaction between the various components like client, server and interface under different scenarios like deposit, withdraw or transfer. A recovery algorithm is defined in order to overcome unexpected situations like server crash.

The Chapter 3 focuses on the java based implementation of the online banking application which support multiple clients simultaneously. The various flows in the application like deposit, withdraw and transfer are tested using exhaustive test cases. The performance of the application is analysed in case of single or multiple clients and single server interaction using timestamps. Graphs are plotted to represent the performance variation

.

# Contents

Table of Contents

[Declaration Sheet ii](#_Toc518690089)

[Abstract iii](#_Toc518690090)

[Contents iv](#_Toc518690091)

[List of Figures v](#_Toc518690092)

[List of Tables vi](#_Toc518690093)

[PART A - CHAPTER 1 7](#_Toc518690094)

[1.1 Introduction 7](#_Toc518690095)

[1.2 Merits and Demerits of Cloud and IT Infrastructure 7](#_Toc518690096)

[1.3 Challenges and Considerations 8](#_Toc518690097)

[1.4 Stance Taken with Justification 9](#_Toc518690098)

[1.5 Conclusion 9](#_Toc518690099)

[PART B - CHAPTER 2 10](#_Toc518690100)

[2.1 Introduction 10](#_Toc518690101)

[2.2 Designing of Effective Solution for the Scenario 10](#_Toc518690102)

[2.3 Interaction Sequence Identification and Modelling 12](#_Toc518690103)

[2.4 Algorithm/Flowchart 15](#_Toc518690104)

[2.5 Recovery Algorithm 18](#_Toc518690105)

[2.6 Conclusion 20](#_Toc518690106)

[PART C - CHAPTER 3 21](#_Toc518690107)

[3.1 Introduction 21](#_Toc518690108)

[3.2 Client-Server Banking Application 21](#_Toc518690109)

[3.3 Multiple Clients and Server for Banking Application 23](#_Toc518690110)

[3.4 Testing of banking application 25](#_Toc518690111)

[3.5 Performance Analysis 28](#_Toc518690112)

[3.6 Conclusion 29](#_Toc518690113)

[Benefits of solving the assignment 29](#_Toc518690114)

[References 29](#_Toc518690115)

[APPENDIX A 30](#_Toc518690116)

# List of Figures

[Figure 2.1: Remote Method Invocation 10](#_Toc518687063)

[Figure 2.2: UML Sequence diagram for Login Flow 11](#_Toc518687064)

[Figure 2.3: Sequence diagram for depositing amount into the account 12](#_Toc518687065)

[Figure 2.4: Sequence diagram for withdraw of amount from the account 13](#_Toc518687066)

[Figure 2.5: Sequence diagram for transferring an amount 13](#_Toc518687067)

[Figure 2.6: Flow chart for login flow 14](#_Toc518687068)

[Figure 2.7: Flow chart for deposit flow 15](#_Toc518687069)

[Figure 2.8: Flow chart for withdraw flow 16](#_Toc518687070)

[Figure 2.9: Flow chart for transfer flow 17](#_Toc518687071)

[Figure 3.1: Login failure due to incorrect password 21](#_Toc518687072)

[Figure 3.2: Deposit fail due to invalid amount 21](#_Toc518687073)

[Figure 3.3: Constraints on the amount entered during withdraw 22](#_Toc518687074)

[Figure 3.4: Constraints on the amount entered during transfer of amount 22](#_Toc518687075)

[Figure 3.5: Single server and multiple clients communication 23](#_Toc518687076)

[Figure 3.6: Output for check balance, fetch account number and account name 23](#_Toc518687077)

[Figure 3.7: Deposit flow 23](#_Toc518687078)

[Figure 3.8: The withdraw flow 24](#_Toc518687079)

[Figure 3.9: The transfer flow 24](#_Toc518687080)

[Figure 3.10: Performance Analysis graph 27](#_Toc518687081)

# List of Tables

[[Table 2.1: Transaction table 19](#_Toc509039811)](#_Toc518688261)

[[Table 2.2: Sample entries of the log file 19](#_Toc509039811)](#_Toc518688262)

[[Table 3.1: Test Table 26](#_Toc509039811)](#_Toc518688263)

[[Table 3.2: The Time taken under various scenarios 28](#_Toc509039811)](#_Toc518688264)

# PART A - CHAPTER 1

# Introduction

From the beginning of software industry, organisations have made many investments on the traditional IT infrastructure. Owning the hardware and software resources and their time-to-time maintenance, repair and upgradations require a lot of investment and labour. This is very challenging for small-scale and medium-scale companies. These organisations are always aim to reduce the Total Cost of Ownership (TCO), which requires a lot of time and additional money to hire the consultants to work for the company. However, some of the companies are shifting to cloud based services, which includes lesser cost to enter the market. The start-ups can use the services provided by cloud and apply the customisations required. The cost of the hardware, which would otherwise be too high is reduced by a considerable amount. The burden of maintenance on the companies reduces as the cloud providers take care of the hardware and software maintenance, upgrade and scalability. As a result, the TCO of the company reduces and it can focus more on development, innovation and research.

# Merits and Demerits of Cloud and IT Infrastructure

**Merits of IT infrastructure:**

* It is more secure as no third party can access any of the in-house information.
* Internet is not required to access the information, as it is present within the company.
* One can apply any amount of customisations on the infrastructures without relying on the external support or permission.
* One has physical control over the data and the backup of the data.

**Demerits of IT infrastructure:**

* The initial investment on hardware including its maintenance and upgradation is expensive.
* People have to be employed and trained regularly for the maintenance and update of the soft-wares. Sometimes contractors are also required to maintain the hardware.
* The licence and agreements of the software and the hardware should be updated once in every one or three years.
* Additional space and infrastructure is required to mount the hardware and provide a good suitable environment.
* Small-scale companies are more susceptible to downtime as they have minimum hardware to support secondary systems in case of any failures.
* In case of any disasters, there are more chances of losing data as everything is in-house. Thus keeping the backup of the data offsite reduces data loss during emergencies.

**Merits of Cloud infrastructure:**

* **Low cost:** The initial cost of hardware, upgrading of the hardware and software is very less.
* **Low maintenance:** The companies need not bother about upgrading the hardware and software in every one or three years. They pay the cloud providers based on the use.
* **Scalable:** The company need not worry about scalability of the hardware as the cloud solutions can scale up or scale down based on the business requirements.
* **No single point of failure:** Cloud datacentres support failure recovery mechanism in the form of redundancies and backup thus reducing the load on the company. If one component fails, another component will take over in few milliseconds
* **Increased Storage:** Cloud provides unlimited storage capacity.
* **Increased Availability:** The cloud is not localised to any location. It can be accessed from anywhere and by anyone. This benefits small companies, which are spread across different geographical location.
* **Energy Efficient:** Cloud is developed keeping in mind the energy efficiencies and the environment concerns.

**Demerits of Cloud Infrastructure:**

* **Initial migration Challenges:** The migration of the applications and the data includes a lot of challenges and considerations.
* **Dependency on Internet:** If internet is down on the users or cloud side, its services cannot be accessed.
* Sometimes the long-time costs of the cloud is almost same as having a private server.
* **Change is Government Laws:**  Sometimes the laws regarding the privacy of the data may change and the companies have to take back the data from cloud to maintain them in the local servers.

# Challenges and Considerations

**Considerations while migrating into cloud infrastructure:**

* To identify the degree by which the application has to be changed like virtualization, application migration or application refactoring.
* **Data Management:** A plan for proper backup and archival of data is required such that the dynamic data is close to server and static data is close to the user.
* **Replatforming:** To be prepared to face platform compatibility issues.
* **Security:** To check the security implementation within the application with respect to the encryption in each layer so that the data stored in cloud will not be affected by new sources of threats.
* **Integration:** If there are plans of keeping some of the applications in-house and migrating only few applications, the integration is required.
* **Licencing:** Verify whether the currently used software or tools are compatible with the cloud based licencing.
* **Automation:** Check for automation techniques.

**Challenges of migrating into cloud infrastructure:**

* **Time Factor:** The migration to cloud takes a considerable amount of time.
* **Shortage of Technical Expertise:** Choosing the right “Cloud Service Model” is a major step. Without proper technical expertise, outsourcing of the migration project may be required.
* **Data Loss:** Companies should take care of the data loss which might occur during migration.
* **Cost Estimation:** Many companies face challenge in estimate the cost of migration.
* **People:** After the migration process, it takes time for the people to get used to the cloud infrastructure. This process becomes easy in new companies where people who are well aware of cloud are employed.
* **Security:** Some companies are not sure of security in the new cloud services and thus prefer to keep the data in-house.
* **Infrastructure:** The companies should check the compatibility of the existing infrastructure before migrating it which otherwise would become an obstacle in migration.

# Stance Taken with Justification

**Debate**: ***"Cloud infrastructure will completely replace the traditional IT infrastructure as an effective IT solution."***

**Stance Taken:** The cloud infrastructure will not completely replace the traditional IT infrastructure soon.

There are many factors because of which the companies are postponing the migration of the traditional IT infrastructure to cloud infrastructure. Some of them include: The migration takes a considerable amount of time. There is also a cost involved in it. The migration requires people with technical expertise in cloud based infrastructure to decide on the exact “Cloud Service Model”. In such conditions, the project requires outsourcing or hiring of contractors. When there is an internal IT team for the company, they know more about the unique and specialised hardware and software requirements. Outsourcing the solution may lead to loss of advantage of the personal IT team.

Another factor is that of security. Keeping the data in the cloud may cause more data loss and data breeches through malicious deletion or disasters. There is more chances of an attacker to hijack the traffic or the account through eavesdropping on the user’s activities. Insecure or Improper implementation of interfaces or API’s in cloud may make the provider vulnerable for the attacks. Sometimes the user might have to share the personal data like credentials to use the cloud service. Therefore, many companies prefer to keep their data in-house.

The cloud architecture is still in its growing phase considering which it will take few years for it to replace the existing systems. Though the usage of cloud is increasing at a faster pace, there are still many use cases, which are specific to the traditional architecture and it will continue to exist even in the future. Majority of the companies who want to adopt cloud will not completely replace their existing systems. In turn, they opt for hybrid structure, which is a combination of cloud and traditional infrastructure. Thus, we can conclude that the cloud will not completely replace the transitional systems in any near future.

# Conclusion

In the future, every company expects for a balance between the in-house and cloud infrastructure. Not all the organisations will replace their complete traditional infrastructure due to their unique and specific requirements and security. Though many of them think that it is easy to replace the entire in-house architecture, it is way more complex than that in terms of the considerations and the challenges faced. The companies will adopt cloud based on their considerations like cost, benefits, data security, people, time, integration. Overall, three scenarios are possible like complete, mixed or non-adoption of the cloud. Anyway, one cannot deny that cloud will become a major technology soon reducing the demand for IT infrastructure. The Research firm 451 Group says that cloud providers such as [Amazon Web Services](https://www.techopedia.com/definition/26426/amazon-web-services-aws) are going to grow at an exponential rate. But, even in the face of growing cloud adoption, traditional architecture is not going to go away anytime soon.

# PART B - CHAPTER 2

# 2.1 Introduction

Servers are those components, which takes requests from client, and provide information or services to them. Online banking application involves such an interaction where an individual uses the browser to contact the server to obtain various services related to his bank account. In this section, a server client interaction in case of online banking application is considered. The user can use the online banking application to get information related to his bank account (E.g. account name, account number, balance) and perform certain actions over them like deposit, withdraw and transfer. The various design considerations, scenarios related to each flow and their related sequence diagram and flow charts are included. In order to combat any failure scenarios, a failure recovery mechanism is also designed.

# 2.2 Designing of Effective Solution for the Scenario

**Given Scenario:** Consider an online client-server based banking application, where customers perform transactions like balance enquiry, deposit and money transfer. A single server maintains all the accounts of customers. Multiple clients can access the server at the same time to invoke the above transactions. Design and implement the banking application so that all the concurrent transactions invoked by all the customers reflect correct and consistent states of account balance in all customers’ accounts.

Designing of an effective solution for the above scenario consist of identifying the best technique to implement the client-server interaction, which can support multiple clients. There are two designing techniques to implement single server, multiple client scenario. They include:

* **Client Server threading for multiple clients**

In this technique, when a client connects to the server, the server spawns a new thread to handle the client. This new thread acts like a child server, which manages the further interaction with that client. Every time a new client joins, a thread is created. The individual request reply between the client and the child server is in the form of input and output stream. The serialised data is passed from the client to server and back to client. This process is very complicated, as the only form of communication is byte stream. The client cannot invoke any specific functionality in the server, in turn the server invokes the required functionality based on the data from the client.

* **Remote Method Invocation (RMI)**

In this technique, the client can directly invoke the required method in the remote server object. It consists of various components like the communication module, which manages the request and reply, remote reference module, servant, dispatcher, proxy, remote interface and skeleton. The client invokes the methods exposed by the proxy. The proxy server then uses the remote object reference to invoke the specific method in the remote interface. It also receives the reply from the remote method and sends it back to the client. The Figure 2.1 shows a sample for the RMI technique.

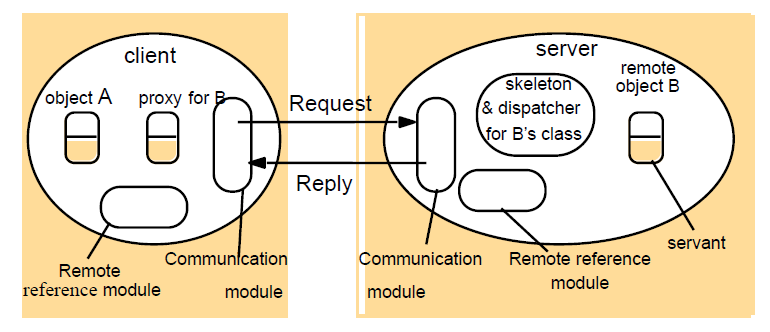


Figure 2.1: Remote Method Invocation

**Design:**

In this assignment, the RMI technique is used to implement the client-server based online banking application. Three components are used to implement the scenario. They include:

* **RMI Client:** The class **BankRMIClient** is used as the RMI client. The functions performed by the client include:
* It manages the login of the user to the online banking application
* Displays the various services provided by the application to the user
* Takes the input from the user regarding the service required
* Prepares the basic data required to perform the service selected
* Calls the corresponding RMI interface method (deposit, transfer or withdraw )and passes the prepared data as the argument.
* Receives the data returned by the RMI interface
* Formats the data
* Displays the data on the console for the user to view
* **RMI Interface:** The Remote Interface helps in hiding the user data, which is confidential, and the implementations of the various functionalities performed by the server. It exposes only the abstract methods, which can be called by the client directly based on the service requested by the user. The interface does not have any access to the user’s data. Thus, the client has no direct access to the user data or the implementation of the method. When the client invokes the interface method, the implementation provided in the server class is executed. The class **BankRMIinterface** represents the RMI interface containing abstract methods for checking account details, deposit, withdraw and transfer money.
* **RMI Sever:** The BankRMIServer represents the server. The server has direct access to all the data of the user. It implements the RMI interface thus providing the implementation of all the abstract methods in the BankRMIInterface. When the client invokes the interface method, the implementation in the server is executed where the server access the required data and updates the existing data if required and returns the final balance to the client.

**Data Management**

If we assume that when multiple clients login and access only their own data, there is no need of applying synchronisation techniques like locks on the data. But, when functions like transfer are performed, then the data of another user need to be modified. By any chance if the other user is also logged in and trying to access or manipulate his data, there might be corruption of data. Under this scenario, the locking mechanism can be used to lock and unlock the data while accessing it.

# 2.3 Interaction Sequence Identification and Modelling

The designing of the online banking application consists of multiple flows like:

* Login to the application
* Check balance
* Fetch account number
* Fetch account name
* Deposit some amount
* Withdraw of some amount
* Transfer some amount to another account

In this section, sequence diagrams designed for four major flows, which include login, deposit, and withdraw. Four components are considered: User, RMI Client, RMI interface and RMI Server. The sequence diagram shows the series of interactions between these components. The sequence diagram for each flow is as follows:

**Login Flow**

The Figure 2.2 shows the sequence diagram for the login flow.

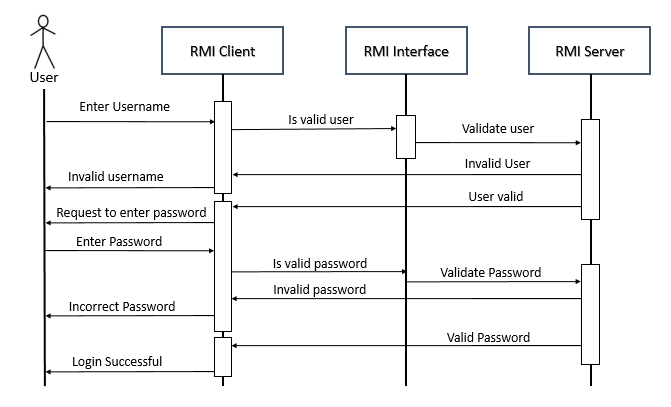


Figure 2.2: UML Sequence diagram for Login Flow

Here the user or the customer interacts with the browser, which acts as the RMI client. The RMI client makes remote method calls to the RMI server with the help of RMI interface. In the login flow, the user first enters the username. The server checks for the presence of the username in the repository. If the username is valid, the user enters the password. If the password is correct, the login is successful and the user can go forward with the further operations on his account. If the password is not correct, the user needs to login again.

**Deposit Flow**

The Figure 2.3 shows the UML sequence diagram for the deposit flow. Once the user has logged in successfully, if he selects the option for depositing some amount, the client asks him to enter the amount to deposit. Once the user enters the amount, the client carries out the first round of validation to check that the amount entered is non-negative. If the amount is non-negative, the client makes a call to the remote interface to deposit the amount followed by the execution of the implementation of the deposit. Once the balance is updated, the user is notified that the deposit was successful.

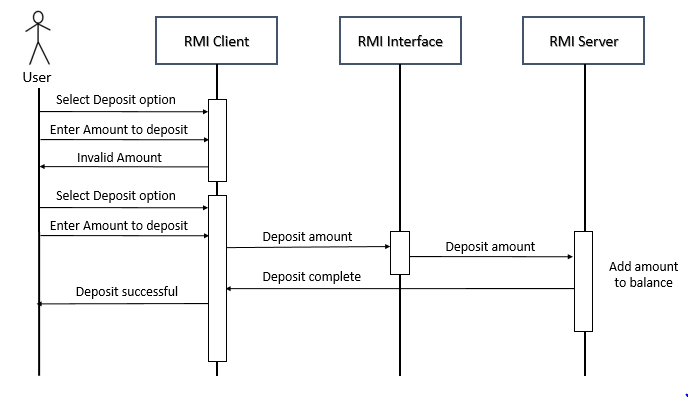


Figure 2.3: Sequence diagram for depositing amount into the account

**Withdraw Flow**

The Figure 2.4 shows the sequence diagram for withdrawing some amount from the account. Once the user has logged-in successfully, he selects the withdraw option to withdraw some amount from his account. The client asks the user to enter the amount he wants to withdraw. Once the user enters the amount, the client carries out the first round of validation to check whether the amount is non-negative. If the amount is not valid, the transaction ends and the user has to select the option again. If the amount is valid, the client calls the withdraw method of the remote interface. The server performs a second round of validation to check that there is enough balance to withdraw. If there is enough balance, the amount is deducted otherwise the transaction is cancelled and the user has to initiate a new withdrawal transaction.

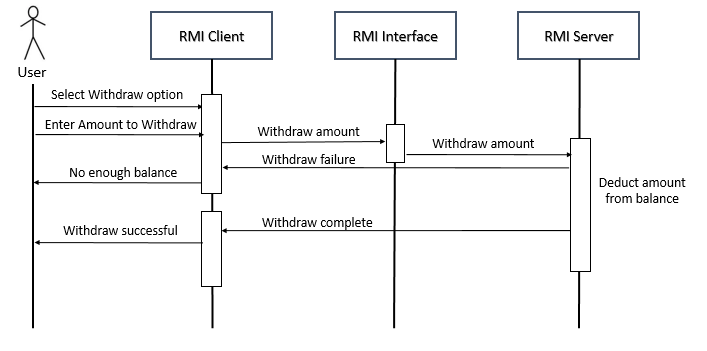


Figure 2.4: Sequence diagram for withdraw of amount from the account

**Transfer Flow**

The Figure 2.5 shows the transfer flow. Here a user defined amount is transferred from the logged in account to the selected account. Once the user selects the option to transfer an amount to another account, he needs to enter the account number to which he is intending to send the amount. Once he enters the account number, the server checks whether the account number is a valid one. If the account number is valid, the client asks the user to enter the amount to be transferred. If the amount is non-negative and the user has enough balance in his account, the transfer is carried out and the defined amount is deducted from his account. If there is no enough balance, the transaction is restarted.

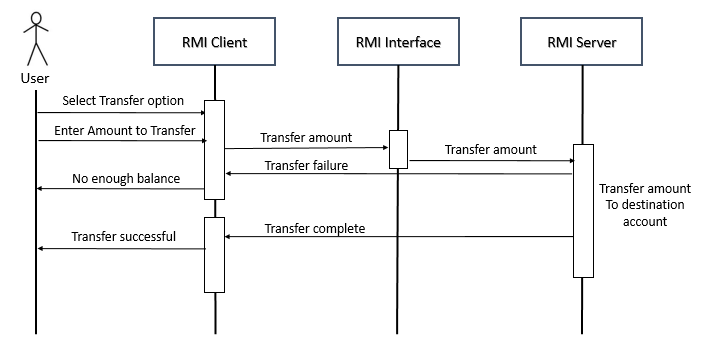


Figure 2.5: Sequence diagram for transferring an amount

# 2.4 Algorithm/Flowchart

A flow chart represents the flow of any program. In a flowchart, rectangles are used to represent the functionalities and diamond shapes are used to indicate the conditional branches. In this section, flow charts are used to represent the different flows in the program which include: login, deposit, withdraw and transfer.

**Flow chart for Login Flow**

The figure 2.6 shows the login flow. Here once the customer enters the username, it is checked for its validity. If the username is valid, the customer enters the password. If the username or password are incorrect, the program comes to an end and the client needs to connect to the server again. If the username and password are correct, the customer is logged-in and he can do the further operations related to his account like deposit, withdrawal, transfer or exit. If the customer chooses exit, he customer is logged out of his account.

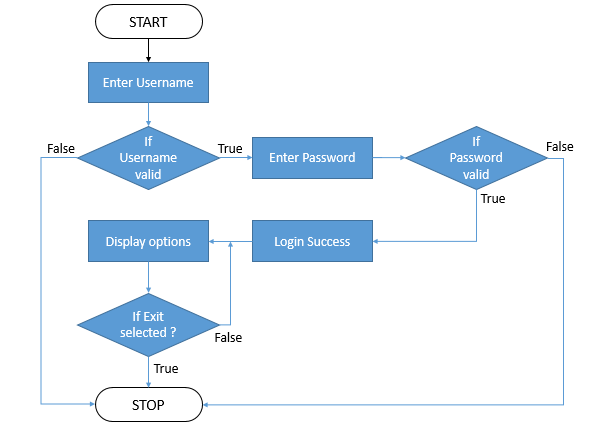


Figure 2.6: Flow chart for login flow

**Flow chart for deposit flow**

Once the customer logs in to his account, various options are displayed to fetch account details, deposit amount, withdraw amount and transfer amount. Once the customer selects the deposit option, he is asked to enter the amount he wants to deposit. Once he enters the amount, it is checked whether the amount is non-negative. If the amount is negative or zero, the transaction comes to an end and the options are displayed again. If the amount is valid, the amount is added to the account balance and the final balance is notified to the customer.

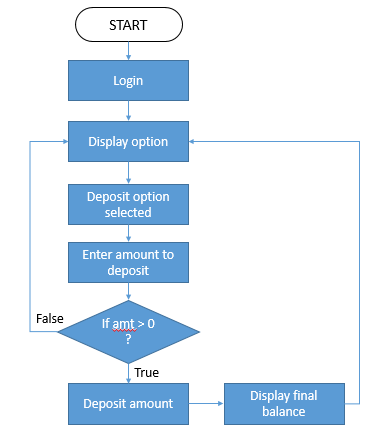


Figure 2.7: Flow chart for deposit flow

**Flow chart for withdraw flow**

Once the customer is logged and he selects the withdraw option, he is asked to enter the amount he wants to withdraw. Once he enters the amount, the program checks the amount for its validity. If the amount is negative, zero or is greater than the existing balance, the customer is not allowed to withdraw the money and the transaction comes to an end. If the amount is valid, the amount is deducted from the account balance and the customer is notified about the updated balance.

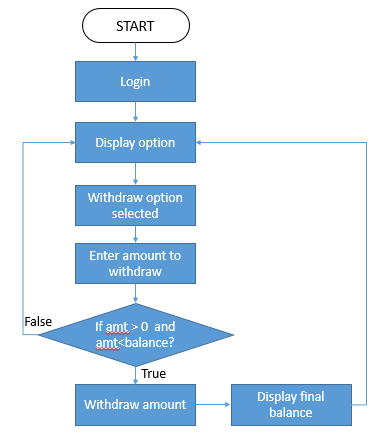


Figure 2.8: Flow chart for withdraw flow

**Flow chart for transfer flow**

Once the customer is logged-in, if he wants to transfer an amount to any other account, he selects the transfer option. Once he selects transfer, he is asked to enter the account number of the account to which he wants to transfer the amount. Then the program checks the account number for validity. If the account number is valid, he is asked to enter the amount he wants to transfer. If the amount is not negative or zero and if the amount is lesser than is current balance, the transfer of the amount occurs. The amount is deducted from the customer’s account and deposited into the destination account. If the customer does not have enough balance, then the transaction comes to an end and the customer has to restart the transaction.

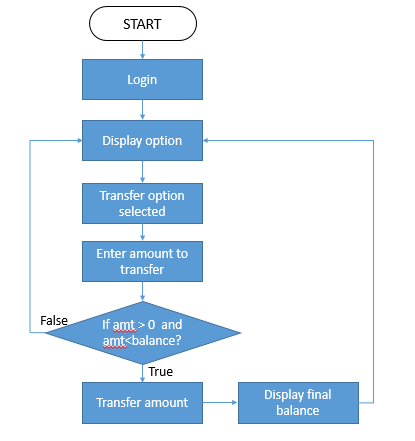


Figure 2.9: Flow chart for transfer flow

# 2.5 Recovery Algorithm

There are two basic kinds of recovery algorithms to overcome scenarios like server crash. They are **logging** and **shadowing.** In this section, Logging recovery method is considered.

In logging, the object values, transaction status and its intention lists are added to a log file in the same order as the various transactions occur. Thus, the log file maintains the history of the transactions. There are three possible status of the transactions, **committed, aborted** and **prepared to commit.** When any server crash occurs, the following steps happen in the recovery process:

* The server is restarted
* The objects are assigned initial default values
* Control is given to recover manager
* Recovery manager starts checking the log file from backward direction
* It check the most recent object values and assigns it to the object
* Checks the last transaction and its intention list
* If the transaction status is not committed, the values of the objects in its intention list are not considered.
* If the transaction status is committed, the values of the objects in its intention list are considered.

Consider an example scenario shown in the Table 2.1 where a deposit transaction (D) and withdraw transaction (W) is performed on the account a1. Let the balance in the account a1 be 100. The execution of the transaction is as follows:

Table 2.1: Transaction table

|  |  |
| --- | --- |
| **Deposit Transaction D** | **Withdraw Transaction W** |
| balance = a1.getBalance() | balance = a1.getBalance() |
| balance = balance + 10 | balance = balance – 20 |
| a1.setBalance(balance) | a1.setBalance(balance) |
| balance = a1.getBalance() //100 |  |
| balance = balance + 10 |  |
| a1.setBalance(balance) //110 |  |
|  | balance = a1.getBalance() //110 |
|  | balance = balance – 20 |
|  | a1.setBalance(balance) //90 |

Table 2.2 shows the entries of the log file when transaction D is committed and transaction W is prepared to commit.

Table 2.2: Sample entries of the log file

|  |  |  |
| --- | --- | --- |
| 1. | a1 | 100 |
| 2. | a1 | 110 |
| 3. | D | Prepared to commit |
| 4. | D | a1 |
| 5. | D | committed |
| 6. | a1 | 110 |
| 7. | a1 | 90 |
| 8. | D | Prepared to commit |
| 9. | D | a1 |

* In the table, the row 1, 2, 6 and 7 show the object values as the transaction occurs
* The row 3, 5 and 8 shows the transaction status of the two transaction considered
* The rows 4 and 9 shows the intention list of the two transactions.

When a crash happens at this stage, the recovery manager scans the log file and takes the object value just before the last commit which occurred. Thus, the balance of the account a1 will be initialised to 110.0, which is the last stable value of the object.

# 2.6 Conclusion

# PART C - CHAPTER 3

# 3.1 Introduction

In this section, the online banking client-server interaction is implemented using java program. RMI technique is used to allow the client to invoke the methods in the remote server directly. The classes required are designed and implemented. Exhaustive test cases are written and the online banking application is tested accordingly. The time taken for each execution is calculated for one client and server communication and multiple client and server communication. Their performances are compared using graphs.

# 3.2 Client-Server Banking Application

The Client-Server Banking Application implemented in java consists of three components: RMI Client, RMI Interface and RMI Server. The Classes used and the methods included are

1. **BankRMIClient** **class**: The client class contains the main method which interacts with the customer on one side and BankRMIInterface class. The functionalities provided by the client are

* Receives the username and password from the customer for login
* Validates the username and password
* Displays the options for the user to select

Choose the operation to perform:

1. Check Balance

2. Display account number

3. Display Account name

4. Deposit

5. Withdraw

6. Transfer

7. Exit

* Based on the option chosen by the user, the client calls the method of the interface.
* It also performs validation of the data entered by the user.

1. **BankRMIInterface class:** It has the following abstract methods

* **validateUser()**
* **validateUserPass()**
* **checkBalance()**
* **fetchAccountNumber()**
* **fetchAccountName()**
* **deposit()**
* **withdraw()**
* **checkValidAccount()**
* **transfer()**

1. **BankRMIServer class:** It consists of the implementation for all the abstract methods present in the interface class. The method implementations include:

* **validateUser() :** checks whether the username provided by the user is valid.
* **validateUserPass() :** Check whether the password is correct for the given valid user.
* **checkBalance() :** Fetches the balance of the account.
* **fetchAccountNumber() :** Fetches the account number of the logged in user.
* **fetchAccountName() :** Fetches the name of the logged in user account.
* **deposit() :** It deposits the user given amount into the account. Thus it adds the given amount to the current balance of the account.
* **withdraw() :** It validates the amount entered by the user. It checks if the amount to be withdrawn is lesser than the balance of the account to which the user has logged in.
* **checkValidAccount() :** This method is used during amount transfer. When the user selects transfer option, he enters the account number to which he wants to transfer the amount. This method checks whether the account number provided by the customer is a valid one.
* **transfer() :** This method performs the transfer of amount from the logged in account to the account specified by the user.

1. **Account class:** This class defines the properties of the account like:

* **accountName**
* **accountNumber**
* **balance**

1. **Bank class:** This class behaves like the data base. It builds the accounts and maintain a list of those accounts.

The various constraints considered in the given scenario and their corresponding verification include:

* The username and password should be valid

The screen shot in the figure 3.1 shows that if a wrong username or password is used, the user is not allowed to login.

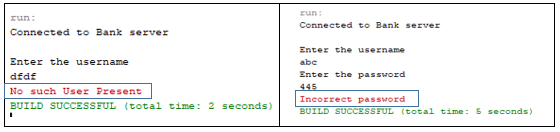


Figure 3.1: Login failure due to incorrect password

* The amount to be deposited should not be negative or zero

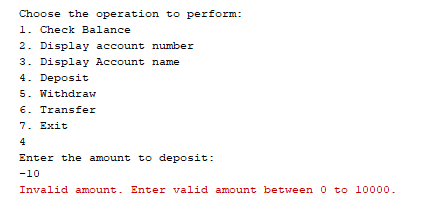


Figure 3.2: Deposit fail due to invalid amount

* The amount to be withdrawn should not be negative, zero or greater than the existing balance. The figure 3.3 shows two transaction failure conditions. One where a negative amount is entered and one when the amount to withdraw is greater than the balance.

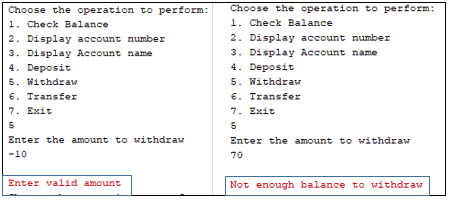


Figure 3.3: Constraints on the amount entered during withdraw

* The amount to transfer should not be negative, zero or greater than the balance. The Figure 3.4 shows the error conditions which occur when the amount entered to transfer is negative or greater than the balance.

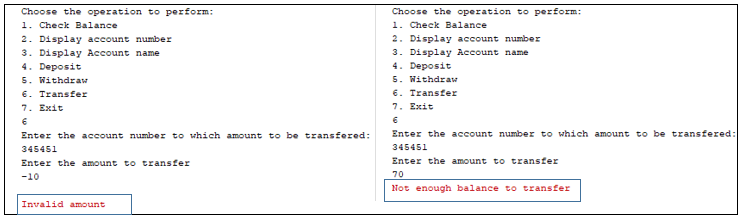


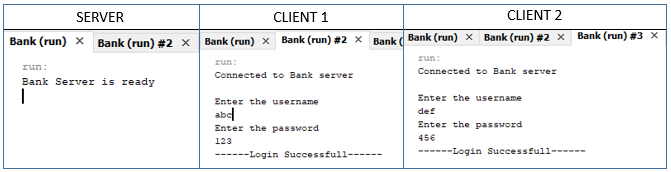
Figure 3.4: Constraints on the amount entered during transfer of amount

# 3.3 Multiple Clients and Server for Banking Application

The code for the application is present in the **Appendix A**.

The various execution flows of the application is as follows:

* Multiple clients can connect to the server simultaneously. The Figure 3.5 shows that there is a single server and there are three other clients who login to the system simultaneously.



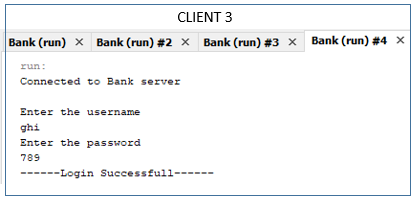


Figure 3.5: Single server and multiple clients communication

* The Figure 3.6 shows the flow to check the balance, account number and account name.

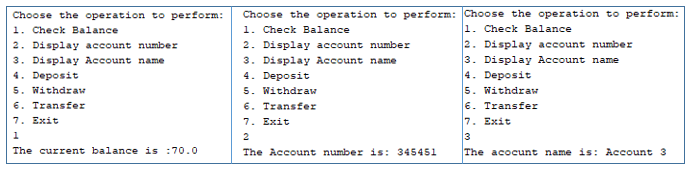


Figure 3.6: Output for check balance, fetch account number and account name

* The Figure 3.7 shows the deposit flow

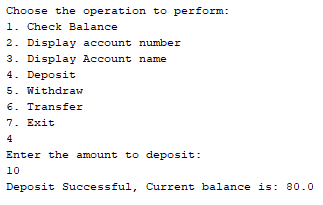


Figure 3.7: Deposit flow

* The Figure 3.8 shows the withdraw flow

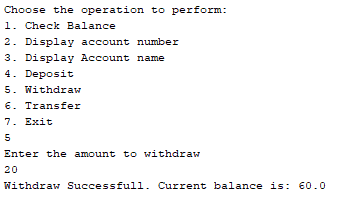


Figure 3.8: The withdraw flow

* The Figure 3.9 shows the transfer flow

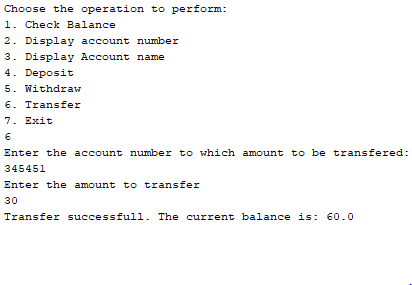


Figure 3.9: The transfer flow

# 3.4 Testing of banking application

Manual tests were carried out on the implementation of the Banking Application with multiple Clients. The Table 3.1 lists all the test cases. Certain initial conditions are considered before starting with the test cases. They include:

* Three accounts are created and details like account name, account number, initial balance, username and password are set. The details for each account include:

1. Account name : Account 1

Account number: 343454

Account balance: 50

User name: abc

Password: 123

1. Account name : Account 2

Account number: 348394

Account balance: 60

User name: def

Password: 456

1. Account name : Account 3

Account number: 345451

Account balance: 70

User name: ghi

Password: 789

Table 3.1: Test Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No | Test Inputs | Test Description | Expected Output | Actual Output | Test Result |
| Login Flow | | | | | |
| 1. | Username: xyz | Username is not present in the bank repository | Login failure due to invalid username | **No such User Present** | Pass |
| 2. | Username: abc  Password: 567 | Password entered is not correct for the username | Login failure due to invalid password | **Incorrect password** | Pass |
| 3. | Username: abc  Password: 123 | Username and password entered are correct | Login success with correct username and password. Display options for further transactions | ------Login Successfull------  Choose the operation to perform:  1. Check Balance  2. Display account number  3. Display Account name  4. Deposit  5. Withdraw  6. Transfer  7. Exit | Pass |
| Fetch account details | | | | | |
| 4. | Option: 1 | To fetch the account balance | Displays account balance: 50 | The current balance is :50.0 | Pass |
| 5. | Option: 2 | To fetch the account number | Displays account number: 343454 | The Account number is: 343454 | Pass |
| 6. | Option: 3 | To fetch the account name | Displays account name: Account 1 | The account name is: Account 1 | Pass |
| Deposit Flow for Account 1 | | | | | |
| 7. | Option: 4  Amount to deposit: -10 | Deposit a negative value amount | The amount is invalid | **Invalid amount. Enter valid amount between 0 to 10000.** | Pass |
| 8. | Option: 4  Amount to deposit: 0 | Deposit a zero value amount | The amount is invalid | **Invalid amount. Enter valid amount between 0 to 10000.** | Pass |
| 9. | Option: 4  Amount to deposit : 10 | A positive amount is deposited | Deposit is successful. Current balance is 60 | Deposit Successful, Current balance is: 60.0 | Pass |
| 10. | Option: 4  Amount to deposit: 10001 | A positive amount greater than 10,000 is deposited | The amount is exceeding the limit. Deposit fail. | **Invalid amount. Enter valid amount between 0 to 10000.** | Pass |
| 11. | Option: 4  Amount to deposit: 45.2 | A decimal value greater than 0 is deposited | Deposit is successful. The current balance is 95.2 | Deposit Successful, Current balance is: 95.2 | Pass |
| Withdraw Flow for Account 1 | | | | | |
| 12. | Option: 5  Amount to withdraw: -10 | Withdraw of negative value amount | Withdraw failure. Amount is invalid | **Enter valid amount** | Pass |
| 13. | Option: 5  Amount to withdraw: 0 | Withdraw of zero value amount | Withdraw failure. Amount is invalid | **Enter valid amount** | Pass |
| 14. | Option: 5  Amount to withdraw: 80 | Withdraw of amount greater than the balance | Withdraw failure. No enough balance to withdraw. | **Not enough balance to withdraw** | Pass |
| 15. | Option: 5  Amount to withdraw: 50 | Withdraw of amount equal to balance | Withdraw successful. Current balance is 0 | Withdraw Successful. Current balance is: 0.0 | Pass |
| 16. | Option: 5  Amount to withdraw: 8 | Withdraw of amount lesser than balance | Withdraw is successful. Current balance is 42 | Withdraw Successful. Current balance is: 42.0 | Pass |
| Transfer Flow from Account 1 to Account 2 | | | | | |
| 17. | Option: 6  To account number: 458612 | To account number does not exist | To account number is invalid | **The account number is invalid** | Pass |
| 18. | Option: 6  To account number: 348394  Amount to deposit: 0 or -10 | To Account number is valid but transfer amount is zero or negative | Transfer amount is invalid | **Invalid amount** | Pass |
| 19. | Option: 6  To account number: 348394  Option: 6  Amount to deposit: 70 | To Account number is valid but transfer amount is greater than the from account balance | Transfer amount is large. No enough balance to transfer. | **Not enough balance to transfer** | Pass |
| 20. | Option: 6  To account number: 348394  Amount to deposit: 10 | To Account number is valid and transfer amount is lesser than the from account balance | Transfer successful. The current balance of the account is 40 | Transfer successful. The current balance is: 40.0 | Pass |
| 21. | Option 7 | To log out of the application | User logged out successfully | ----Customer successfully logged out !!!---- | Pass |

# 3.5 Performance Analysis

In the application considered, there are three main transactions: Deposit, withdraw and transfer.

The application is designed in such a way that it can handle multiple clients simultaneously. In order to analyse the performance of the application in the presence of single or multiple clients, time stamps are used. Just before any of the three transaction starts, the time stamp is recorded known as **start time** and just after the transaction complete, the time stamp is noted again called the **end time.** Finally the difference between the end time and the start time is calculated to know the amount of time taken to execute the functionality. The Table 3.1 shows three conditions shown in the columns 2, 3 and 4. They represent the number of clients connected to the server at a given time. The 2nd, 3rd and 4th row show the execution time for deposit, withdrawal and transfer in case of the 3 conditions.

Table 3.2: The Time taken under various scenarios

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 client | 2 clients | 3 clients |
| Deposit | 710981 | 1201658 | 611399 |
| Withdrawal | 1499923 | 743347 | 1568104 |
| Transfer | 4721313 | 3746161 | 4696825 |

Figure 3.10: Performance Analysis graph

Using the Table 3.1, a graph can be plotted to check the variation in the execution time under the three conditions.

**Analysis:** From the graph shown in the Figure 3.10, we can conclude that

* The single client and server interaction gives stable execution times
* The variation in the execution time is very high form 2 and 3 clients scenario
* Highest Execution times are obtained when there are multiple clients
* But, the difference in the execution times between the single and multiple clients is less which can be ignored. Thus, we can conclude that the application is performing well even in the presence of multiple clients

# 3.6 Conclusion

# Benefits of solving the assignment

# References

Boulanger, A. (2005). Open­source versus proprietary software: Is one more reliable and secure than the other?. IBM Systems Journal, 44(2), 239­248.

Avižienis, A., Laprie, J. ­., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1(1), 11­33.

SANS Institute InfoSec Reading Room(2003) *Security Concerns in Using Open Source Software for Enterprise Requirements* Sreenivasa Rao Vadalasetty, GSEC Practical Assignment - Version 1.4b Option 1, October 15, 2003

The Government of the Hong Kong Special Administrative Region *Open Source Security,* February 2008.

Prashanth Phatak, *Top 10 Security Assessment Tools*(2012) Available at: <https://opensourceforu.com/2012/02/top-10-security-assessment-tools/> ( Last updated: 22nd Feb 2012, Accessed: 31st May 2018)

*Five best practices for open source security* (2007) Available at: <https://searchcio.techtarget.com/tip/Five-best-practices-for-open-source-security> (Last updated: September 2007, Accessed 31st May 2018)

3 Reasons Why Open Source Software is More Secure than Commercial Software. An Opinion (2015) Ron Rymon , Available at : <https://resources.whitesourcesoftware.com/blog-whitesource/3-reasons-why-open-source-is-safer-than-commercial-software-an-opinion>, (Last updated: Apri 7, 2015 Accessed: 1st June 2018)

# APPENDIX A

1. **Account.java**

public class Account {

private String accName;

private String accNumber;

private double balance;

public String getAccName() {

return accName;

}

public String getAccNumber() {

return accNumber;

}

public double getBalance() {

return balance;

}

public void setAccName(String accName) {

this.accName = accName;

}

public void setAccNumber(String accNumber) {

this.accNumber = accNumber;

}

public void setBalance(double balance) {

this.balance = balance;

}

}

1. **Bank.java**

import java.util.HashMap;

import java.util.Map;

public class Bank {

public Map userCred(){

Map<String, String> credentials = new HashMap<>();

credentials.put("abc", "123");

credentials.put("def", "456");

credentials.put("ghi", "789");

return credentials;

}

/\*\*

\* Creating three accounts and adding them on to a map

\* @return

\*/

public Map buildAccounts(){

Map<String, Account> accountsMap = new HashMap<>();

Account acc1 = new Account();

acc1.setAccName("Account 1");

acc1.setAccNumber("343454");

acc1.setBalance(50);

Account acc2 = new Account();

acc2.setAccName("Account 2");

acc2.setAccNumber("348394");

acc2.setBalance(60);

Account acc3 = new Account();

acc3.setAccName("Account 3");

acc3.setAccNumber("345451");

acc3.setBalance(70);

accountsMap.put("abc", acc1);

accountsMap.put("def", acc2);

accountsMap.put("ghi", acc3);

return accountsMap;

}

}

1. **BankRMIClient.java**

import java.rmi.AccessException;

import java.rmi.NotBoundException;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.util.Scanner;

/\*\*

\* @author jhansivsetty

\* 1. incorrect username

\* 2. incorrect password

\* 3. login successfull

\*/

public class BankRMIClient {

public static void main(String[] args) throws Exception {

BankRMIinterface bri = null;

try {

//Registry registry = LocateRegistry.getRegistry("Localhost", 4444);

try {

Registry registry = LocateRegistry.getRegistry("Localhost", 4444);

bri = (BankRMIinterface)registry.lookup("hi server");

System.out.println("Connected to Bank server");

System.out.println("\nEnter the username");

Scanner sc = new Scanner(System.in);

String userName = sc.next();

int option = 0;

double balance,amount;

boolean isValidUser = bri.validateUser(userName);

if(isValidUser){

System.out.println("Enter the password");

String password = sc.next();

boolean isValidUserPass = bri.validateUserPass(userName, password);

if(isValidUserPass){

System.out.println("------Login Successfull------");

while(option != 7){

System.out.println("\nChoose the operation to perform:");

System.out.println("1. Check Balance");

System.out.println("2. Display account number");

System.out.println("3. Display Account name");

System.out.println("4. Deposit");

System.out.println("5. Withdraw");

System.out.println("6. Transfer");

System.out.println("7. Exit");

option = sc.nextInt();

if(option >7 || option<1){

System.err.println("Invalid option\n");

}

switch(option){

case 1: // To check the balance

balance = bri.checkBalance();

System.out.println("The current balance is :"+balance);

break;

case 2: // To get the account number

String accNumber = bri.fetchAccountNumber();

System.out.println("The Account number is: "+accNumber);

break;

case 3: //To get the account name

String accName = bri.fetchAccountName();

System.out.println("The acocunt name is: "+accName);

break;

case 4: //To deposit an amount

System.out.println("Enter the amount to deposit:");

amount = sc.nextDouble();

//System.out.println("The amount is:"+amount);

balance = bri.deposit(amount);

if(balance <0){

System.err.println("Invalid amount. Enter valid amount between 0 to 10000.");

}else{

System.out.println("Deposit Successful, Current balance is: "+ balance);

}

break;

case 5: //To withdraw an amount

System.out.println("Enter the amount to withdraw");

amount = sc.nextDouble();

if(amount<=0){

System.err.println("Enter valid amount");

}else{

balance = bri.withdraw(amount);

if(balance <0){

System.err.println("Not enough balance to withdraw");

}else{

System.out.println("Withdraw Successfull. Current balance is: "+balance);

}

}

break;

case 6: //To transfer an amount

System.out.println("Enter the account number to which amount to be transfered:");

accNumber = sc.next();

boolean isAccPresent = bri.checkValidAccount(accNumber);

if(isAccPresent){

System.out.println("Enter the amount to transfer");

amount = sc.nextDouble();

if(amount<=0){

System.err.println("Invalid amount");

}else{

balance = bri.transfer(amount);

if(balance == -1){

System.err.println("Not enough balance to transfer");

}else{

System.out.println("Transfer successfull. The current balance is: "+balance);

}

}

}else{

System.err.println("The account number is invalid");

}

break;

case 7: break;

default : break;

}

}

System.out.println("----Customer successfully logged out !!!----");

}else{

System.err.println("Incorrect password");

}

}else{

System.err.println("No such User Present");

}

}catch (NotBoundException | AccessException ex) {

ex.printStackTrace();

}

} catch (RemoteException ex) {

ex.printStackTrace();

}

}

}

1. **BankRMIServer.java**

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.rmi.server.UnicastRemoteObject;

import java.util.Map;

import java.util.Set;

public class BankRMIServer extends UnicastRemoteObject implements BankRMIinterface{

private static Map<String,String> userCred;

private static Map<String,Account> accounts;

private Account selectedAccount;

private Account selectedToAccount;

//private Account selectedAccount;

private BankRMIServer() throws RemoteException{

super();

}

public static void main(String[] args) {

try {

Registry registry = LocateRegistry.createRegistry(4444);

registry.rebind("hi server", new BankRMIServer());

System.out.println("Bank Server is ready");

Bank bank = new Bank();

userCred = bank.userCred();

accounts = bank.buildAccounts();

} catch (RemoteException ex) {

ex.printStackTrace();

}

}

@Override

public boolean validateUser(String userName) throws Exception {

return userCred.containsKey(userName);

}

@Override

public boolean validateUserPass(String userName, String password) throws Exception {

String pass = userCred.get(userName);

boolean isValidUserPss = pass.equals(password);

if(isValidUserPss){

selectedAccount = accounts.get(userName);

}

return isValidUserPss;

}

//To check the balance of the user

@Override

public double checkBalance() throws Exception {

return selectedAccount.getBalance();

}

@Override

public String fetchAccountNumber() throws Exception {

return selectedAccount.getAccNumber();

}

@Override

public String fetchAccountName() throws Exception {

return selectedAccount.getAccName();

}

//To deposit amount only if it is between 0 and 10000

@Override

public double deposit(double amount) throws Exception {

if(amount > 0 && amount <= 10000){

double currBalance = selectedAccount.getBalance();

currBalance += amount;

selectedAccount.setBalance(currBalance);

return selectedAccount.getBalance();

}else{

return -1;

}

}

@Override

public double withdraw(double amount) throws Exception {

double currBalance = selectedAccount.getBalance();

if(amount > currBalance){

//in case the amount to withdraw is greater than the balance

return -1;

} else{

currBalance -= amount;

selectedAccount.setBalance(currBalance);

return selectedAccount.getBalance();

}

}

@Override

public boolean checkValidAccount(String accNumber) throws Exception {

boolean isAccountExist = false;

Set<String> keySet = accounts.keySet();

for(String uname:keySet){

Account account = accounts.get(uname);

if(account.getAccNumber().equals(accNumber)){

isAccountExist = true;

selectedToAccount = account;

break;

}

}

return isAccountExist;

}

@Override

public double transfer(double amount) throws Exception {

double currBalance = selectedAccount.getBalance();

System.out.println("Initial balance of from account with account number: "+selectedAccount.getAccNumber()+" is "+currBalance);

System.out.println("Initial balance of to account with account number :"+selectedToAccount.getAccNumber()+" is "+ selectedToAccount.getBalance());

System.out.println("Amount to transfer: "+amount);

if(amount > currBalance){

return -1;

}else{

currBalance -= amount;

selectedAccount.setBalance(currBalance);

System.out.println("The balance of from acc after transfer: "+selectedAccount.getBalance());

double toAccBalance = selectedToAccount.getBalance();

toAccBalance += amount;

selectedToAccount.setBalance(toAccBalance);

System.out.println("The balance of to account after transfer: "+selectedToAccount.getBalance());

}

return selectedAccount.getBalance();

}

}

1. **BankRMIInterface.java**

import java.rmi.Remote;

public interface BankRMIinterface extends Remote {

boolean validateUser(String userName) throws Exception;

boolean validateUserPass(String userName, String password) throws Exception;

double checkBalance() throws Exception;

String fetchAccountNumber() throws Exception;

String fetchAccountName() throws Exception;

double deposit(double amount) throws Exception;

double withdraw(double amount) throws Exception;

boolean checkValidAccount(String accNumber) throws Exception;

double transfer(double amount) throws Exception;

}