

Distributed Banking System

Team Members:

Shivang Raj (15BCE0691)

Saksham (15BCE0279)

Kanav Sethi (15BCE0311)

**Report submitted for the
Final Project Review of**

Course Code: CSE4001 – Parallel and Distributed Computing

Slot: D2

Professor: Prof. Anuradha G

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	3
1.	Introduction	3
	1.1. Synopsis	3
	1.2. Theoretical Background	4
	1.3. Motivation	5
2.	Literature Review	5
3.	Proposed work	12
	3.1. Aim of proposed work	12
	3.2. Problem statements	12
	3.3. Constituents	13
4.	Software Requirements	13
5.	Proposed System World	14
	5.1. JSD	14
	5.2. Problem Analysis Chart	14
6.	Expected Output	15
7.	Implementation	15
	7.1. Methodology	15
	7.2. Code	20
	7.3. Screenshot	26
8.	Conclusion	37
9.	Future Developments Possibilities	37
10.	References	37

ABSTRACT

In this implementation of the distributed banking system we use java to create a modular executable distributed system and each of the above-mentioned modules interact using Remote Method Invocation(RMI). Here multiple clients can access the same server at same time without any collisions. This is reminiscent to virtual client-server system as the data structure will serve similar functionality as that of server storage and a java program is written to manage this data structure whereas the client will be emulated by multiple instances of the windows terminal. The client will first login (necessary for server to authorize the client) and after doing so client will have access to some basic operations like deposit, withdraw, check balance and view transaction history. All the client-side instructions will have to be given in the form of queries written in the command prompt, which the java program of the client side will parse and communicate the request to the server side. The pros for decentralizing the processing conquer the cons so, here we are implementing a distributed banking system that consists of a server and a network of several clients where server maintains and manages all users' account data and the consumers could interact with clients and perform their operations offered by server.

1. Introduction

1.1. Synopsis

The following is the report for my project done at VIT University in Fall semester 2017-2018 session. The project is towards the fulfilment of J-component of four credit Parallel and Distributed Computing course at VIT University. The project has been done in a group of three with a great assist of our faculty assigned to the course. The report is an in-depth presentation of all the things performed throughout the phase of the project. This whole project was a big learning lesson for us.

1.2. Theoretical background

In this day and age of developing advancements, endeavors are moving towards the Internet for trade and business. Individuals are surging towards the web based business applications for their everyday needs, which thus are making the Internet extremely well known. Internet Banking has given both an open door and a test to conventional managing an account. In the quickly developing world, managing an account is a need, which takes a great deal of time from our bustling schedule. Setting off to a branch or ATM or paying bills by paper look at and mailing them, and adjusting checkbooks are untouched expending works. Managing an account online computerizes a significant number of these procedures, sparing time and cash. For all banks, internet saving money is a capable apparatus to increase new clients while it serves to dispenses with exorbitant paper dealing with and manual teller associations in an undeniably focused keeping money condition. Banks have spent ages picking up trust of their clients, and the objective for this venture is to build up an easy to understand, secure Online Banking Application and a reliable source of money transactions for customers. Current and sudden increase in services moving to on demand anywhere access platform can be credited to the boom in the access people have to the internet. This implies that the on the go access of a variety of services is no longer a concept, rather it's a possibility. One such service is the banking sector, as every bank is now extensively pushing towards the goal of providing their customers the freedom of accessing all transaction services online, on demand and securely. To accomplish this, it is necessary for the developers to provide a parallel and user-friendly environment which is able to cope with multiple users accessing the database concurrently because if this is not accomplished the system will be obsolete. Another major concern is making the system robust i.e. if any problem occurs at any point of time during the transaction, the system should rollback the transaction. In this whole process of developing prototype of distributed banking system, we have followed a effective level of abstraction and modulation.

1.3. Motivation

On demand services are experiencing increasing need as more and more people are interested in the possibility of accessing services on the go. So, with this trend in mind we tried to find an application of the course **CSE-4001 Parallel and Distributed Computing** and incorporate it into our project. Since this course deals with parallel execution and the communication between tasks we had the idea of implementing a distributed banking system where the different terminal windows will simulate the independent clients/users which will try to simultaneously access a server emulated by a database and controlled by a java program. Once they are granted the access they can perform some basic transaction operations.

2. Literature Review

Paper [1] sheds light on the basic introduction of the concepts of Distributed system most noticeably the definition and few defining characteristics of the same. After analyzing various sources which often contradict each other this paper defines distributed systems as:

“A distributed system is a collection of independent computers that appears to its users as a single coherent system.”

Analysis of the same renders various important concepts mentioned in it. First and foremost, the components are termed as autonomous rather than dependent, secondly, there must be some interaction between these autonomous units to form a collaboration and this interaction forms a vital and defining characteristic of distributed system i.e. this makes these otherwise autonomous units appear on coherent system to the user which may be a program or a programmer. It is worth mentioning that these autonomous units are under no constraints with respect to their processing power or functionality. Further, an important characteristic of a distributed system is that the way the autonomous units differ, and the way they interact in usually not visible to the user. Another characteristic is that

the interaction between the user and system is independent of where and when it occurs. These characteristics provide distributed system ability to be more scalable and robust. In principle, allocated systems also needs to be not too difficult to extend or level. This feature is a primary consequence of experiencing independent pcs, but at the same time, concealing how these pcs actually be a part of the system all together. A sent-out system will normally be consistently available, although perhaps some parts may be briefly out of order. Users and applications shouldn't observe that parts are being changed or set, or that new parts are put into provide more users or applications. Because you'll be able to build sent out systems will not necessarily mean that it's a good notion. In the end, with current technology additionally it is possible to place four floppy drive drives on an individual computer. It is merely that doing this would be pointless. In such a section we discuss four important goals that needs to be achieved to make creating a distributed system well worth your time and effort. A sent-out system should make resources easy to get at; it should fairly hide the actual fact that resources are allocated across a network; it ought to be open; and it ought to be scalable. Distributed systems contain autonomous pcs that interact to give the looks of an individual coherent system. One important gain is that they make it better to incorporate different applications working on different computer systems into an individual system. Another benefit is that whenever properly designed, allocated systems level well with regards to the size of the actual network. These advantages often come at the price tag on more technical software, degradation of performance, and also often weaker security. Nevertheless, there may be substantial interest worldwide in building and putting in distributed systems. Sent out systems often target at hiding lots of the intricacies related to the syndication of operations, data, and control. However, this syndication transparency not only comes at a performance price, however in sensible situations it can't ever be completely achieved. The actual fact that trade-offs have to be made between obtaining various kinds of syndication transparency is natural to the look of sent out systems, and can certainly complicate their understanding.

- Issues are further complicated by the actual fact that many designers primarily make assumptions about the main network that are fundamentally incorrect. Later, when assumptions are decreased, it may grow to be difficult to cover up unwanted behavior. An average example is let's assume that network latency is not significant. Later, when porting

a preexisting system to a wide-area network, covering latencies may deeply have an impact on the system's original design. Other pitfalls include let's assume that the network is reliable, static, secure, and homogeneous.

Firstly, we needed to understand distributed banking system for which paper [2] proved vital as it is reminiscent of the problem statement of our project. In this paper [2], the author reflects on the changing trend of today's consumer, demanding easier and readily available access of services which forces the service providers to do but in pursuit of designing such service delivery framework various problems arise. Particularly focusing on banking system this paper states that due the introduction of online banking services and ATM and so on the problem of simultaneous access of same data has grown, which is a text book application of distributed system further, distributed system provides the much-required reliability and robustness in these systems. This paper [2] shows, how Distributed applications like online Bank, may easily be developed using Businesses Java Technologies and its own distributed architectures. A far more efficient approach is always to utilize Model View Controller 2 structures, which makes program reusable, robust, plus more object-oriented. J2EE, as allocated component structures, is the right solution in expanding web applications.

Next step was the communication, which was accomplished by Java RMI for which we refer to paper [3] which talks about the functionality of Java in achieving parallel computing. This paper [3] then specifically talks about the Remote Method Invocation (RMI) provided by Java. RMI provides a flexible remote call procedure that supports polymorphism. The paper [3] further talks about the different methods of using the Java RMI and how to implement it efficiently. RMI also lets you download remote classes into the program at runtime. Further they talk about two specific implementations of Java RMI. Firstly, the Manta Java System and the compares it to the Sun RMI implementation. Further the talk about their respective speeds and RMI-null latency time. The main efforts of this paper [3] are as first, we show that RMI can be carried out effectively and can buy a performance near that of RPC systems. The null-RMI latency of Manta RMI over Myrinet is 37 us, only 6 us slower when compared to a C-based RPC standard protocol. Second of all, we show that high performance may be accomplished while still

encouraging polymorphism and interoperability with JVMs by using powerful bytecode compilation and multiple RMI protocols. Next, we provide a detailed performance assessment between your Manta and Sunshine RMI protocols, using benchmarks and a assortment of six parallel applications. To permit a fair comparability, we put together the applications and sunlight RMI process with the indigenous Manta compiler. The results show that the Manta standard protocol ends up with 1.8 to 3.4 times higher speedups for four out of six applications. The rest of the paper [3] is organized the following. Design and execution of the Manta system are reviewed in Section 2. In Section 3, we provide a detailed examination of the communication performance of our bodies. In Section 4, we discuss the performance of several parallel applications. In Section 5 we look at related work. Section 6 presents conclusions. Paper [4] helps us to have better understanding of banking system. The earlier traditional banking systems had only single tier systems, supported centralized data bases and had a large number of limitations. In modern times, the users want and needs have no limits and the access to the data is required to be global and updated regularly. Traditional systems provided only read access to the users. Moderns systems have no such limitations and its client-side computing environment can be available logically on the server globally and can be accessed remotely. In this paper [4], author shown a research point of view on sent out system in medical care sector. From the survey it sometimes appears that, the existing functions of health services are insufficient due to insufficient communication facilities between them. Also today's data recording types of procedures in almost all of health centers leave much to be desired. Manual systems utilized are inaccurate requires time, space and cost. Because of insufficient well processing facilities, unavailability of current data and recent information ,we thought there's a great need of modern communication system that reduces wastage of resources ,time and cost. To be able to minimize these downsides, we must develop sent out system predicated on network technology, logger strategy and online technology. The earth is changing at fast rate which is the need of that time period to hook up all the rural and remote control places by making use of cordless technology.i.e. Internet using cellular phone. Within the last couple of years of the nineteenth hundred years and early couple of years of the twentieth hundred years, medical benefits were quickly employed by method of the progress manufactured in the field of analogue telephony. Individuals,

through this progress, could actually call the physician when in need. Nursing homes also implemented it by transmitting electrocardiograms over mobile phone lines. We were holding the early days and nights of "tele"- remedies or health care delivered remotely.

Paper [5] starts with introducing the DOM model for transaction handling. Further it talks about the various merits and demerits of the DOM model. Due to the demerits of the DOM model a need for a new more powerful transaction model arose. To solve the compatibility issues the new DOM Transaction management system (TMS) was introduced. It classified the models based on type and correctness criteria. Correctness criteria is the degree of concurrency the model has. This separation of transaction model and correctness criteria helps us choose and clarify a large number of transaction models. Two recent trends in data management will be the emergence of databases systems with expanded features [Atkinson et al., 1989; Stonebraker et al., 1990a], and the integration of multiple, heterogeneous data source systems [Litwin, 1988; Gupta, 1989; Sheth and Larson, 1990]. A far more ambitious goal is the integration of autonomous, heterogeneous data source and non-database systems into a sent out processing environment. The DOM (Distributed Subject Management) job at GTE Laboratories addresses specifically these issues. The DOM task has as its goal the introduction of a sent out, object-oriented environment where new (non-traditional) applications can be developed, and where autonomous, heterogeneous systems can be included [Manola, 1988 and 1989; Manola and Buchmann, 1990].

Right concept implementation has the ability to exponentially elevating the performance measures of applications in multicore architectures. The paper [6] deems speculative parallelism as the future trend for general-purpose applications. Newly proposed techniques for parallelization demonstrate performance as scalable on multiple cores. But most software and hardware thread level speculation memory system are not sufficient as they are compatible only to single threaded atomic units. Almost all the methods of exterminating this problem are expensive as they require external expensive hardware. The paper [6] is trying to deal with this problem on software level. With the right techniques, multicore architectures might be able to continue the exponential performance development that increased the performance of applications of most types for many years.

While many medical programs can be parallelized without speculative techniques, speculative parallelism is apparently the main element to carrying on this pattern for general-purpose applications. Recently-proposed code parallelization techniques, such as those by Bridges et al. and by Thies et al., demonstrate scalable performance on multiple cores by using speculation to split code into atomic products (ventures) that course multiple threads to be able to expose data parallelism. However, most software and hardware Thread-Level Speculation (TLS) ram systems and transactional memory aren't sufficient because they only support single-threaded atomic models. Multi-threaded Ventures (MTXs) address this issue, nevertheless they require expensive hardware support as presently suggested in the books. This paper proposes a Software MTX (SMTX) system that catches the applicability and performance of hardware MTX, but on existing multicore machines.

Two queueing network models which are related to separate lock management algorithm implementation aimed at concurrent transaction processing for a database system. Primary aim of development of these models as discussed in paper [7] is to investigate the effects of the degree of multiprogramming and varying the granularity of locks, on the performance of a database. These models adhere to Ries and Stonebraker results and confirms that enough parallelism can be provided by relatively coarse granularity for efficient resource utilization. These models also closely examine cause-effect relationships low cost concurrent transaction processing. Concurrent transaction handling in a repository system is a way of optimizing system performance and learning resource usage by allowing distributed use of data. Several undesired situations might occur if the distributed usage of the data source by concurrent ventures is not properly governed. These unwanted situations include lack of revise, non-repeatable read and restoration with domino impact [4,51. In order to avoid occurrence of the undesirable situations also to preserve the persistence of the data source, concurrency control must be enforced. The correctness criterion for a concurrency control system which preserves database persistence is the capability to ensure that concurrent execution of a couple of transactions is the same as the execution of the same transactions in a few serial order.

The currently common concurrency control in database systems are based data objects constrains for control mechanism. The reference paper [8] provides us with a couple of new theories regarding nonlocking concurrency control. These aforementioned theories though are based assumption that conflicts between transaction doesn't help. Also, transaction backup as a control mechanism is vital point. Consider the challenge of providing distributed usage of a database sorted out as a assortment of objects. We suppose that certain recognized items, called the origins, are always present and usage of any object apart from a main is gained only by first accessing a main and then pursuing pointers compared to that object. Any collection of accesses to the repository that preserves the integrity constraints of the info is named a exchange, If our goal is to increase the throughput of accesses to the data source, then there are in least two instances where highly concurrent gain access to is desirable.

(1) The quantity of data is sufficiently great that at any moment only a small fraction of the repository can be there in primary recollection, such that it is essential to swap elements of the repository from secondary memory space as needed.

(2) Whether or not the whole databases can be there in primary recollection, there could be multiple processors.

In both situations the hardware will be underutilized if the amount of concurrency is too low.

Paper [9] talks about a portable banking system which has on host computer and at least one terminal among other ATM and personal computers. The terminals have minimal keyboard and display functionality. The control and mode of operations reside in the host computer therefore minimal change is required to add new functionality. Also, the cost of providing terminals is very less as compared to personal terminals and example to ATM's as terminals is also talked about in details. A lightweight personal bank operating system comprises a bunch computer with least one lightweight terminal among a number of terminals including computerized teller machines and computers. The lightweight terminal offers a range of bank services over an automatically dialed-up phone link with the host standard bank computer. The non-public terminal includes a computer keyboard and display capabilities with mininal key requirements and screen capacity respectively.

Furthermore, throughout a banking setting of procedure, control resides in the variety bank computer. Subsequently, future changes or improvements to bank services may be executed without the change to the non-public terminal. Also, the expenses of providing today's lightweight personal terminal bank terminals are nominal weighed against known home information terminals such as videotex terminals or computers. Yet, an individual terminal offering system works with with a offering system including office or home personal computers prepared with modems, computerized teller machines and other loan provider service enhancements executed by the providing bank. Also, throughout a local method of procedure minus the bank operating system, the terminal provides as an individual banking manager. The non-public terminal real estate may consist of a prominent surface for data access and screen and a rear end surface composed of a pocket for bank varieties and a writing desk.

All these papers are centered on the basic distributed banking system implantation analysis. It goes with the introduction then the trending scenario all over. And then how could it be dealt with Java using RMI for the implementation of polymorphism. Some of the transaction models has been analyzed. Also, portable banking system has been studied on the basis of the previous analysis.

3. Proposed Work

3.1. Aim of the proposed Work

The aim of this work to create a client-server based program which communicated via Java RMI. Here the client i.e. terminal initiates an operation by calling a remote method on the bank server to execute some basic transaction functionalities but first of all login is required to authenticate the user account for which simple array based data structure has been used.

3.2. Problem Statement

To implement a distributed banking system where the different terminal windows will simulate the independent clients/users which will try to simultaneously access

a server emulated by a database and controlled by a java program. Once they are granted the access they can perform basic transaction operations like deposit, withdrawal, inquiry and bank statement receipt.

3.3. Constituents

This distributed banking system consists of a server and a client. The server all users' basic account information that can be invoked by customer via terminal. The functionalities are as follows:

- **deposit:** in this operation the user account's balance would get increased by the amount entered.
- **withdraw:** using this operation the user account's balance would get deducted by the amount entered.
- **inquiry:** using this operation the user would get to know his/her balance of respective account.
- **getStatement:** this operation returns a statement having transactions over a period of time.
- **login:** this operation is to authenticate the user account accessibility.

Here we have written a client-server based program which communicated via Java RMI. Here the client i.e. terminal initiates an operation by calling a remote method on the bank server to execute one of the above-mentioned functionalities but first of all login is required to authenticate the user account for which simple array based data structure has been used. If the login succeeds a session ID is returned which is then valid for 5 minutes to use. The session ID acts as an authentication token that must be passed for each of the other remote methods.

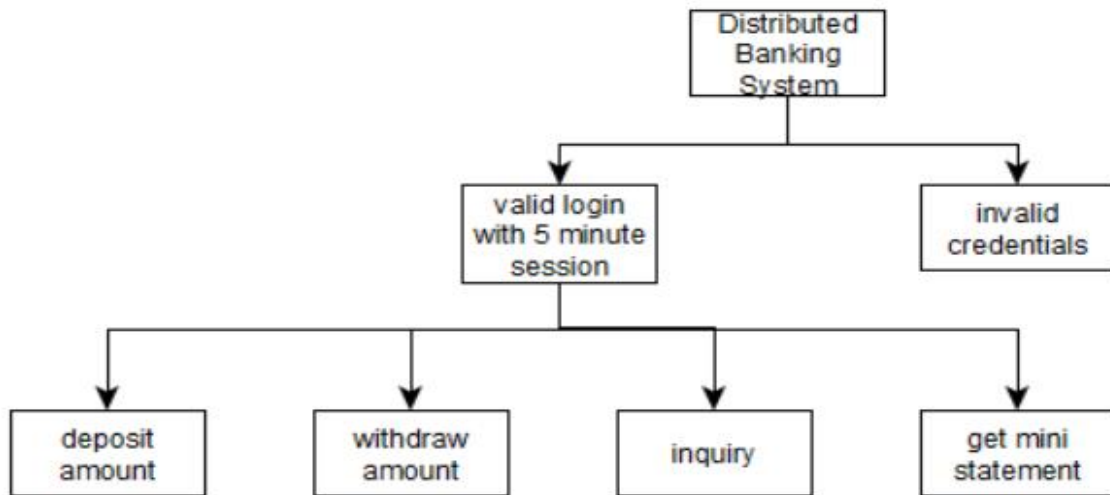
4. Software Requirements

- Eclipse IDE
- JDK (Java Development Kit)

- JRE (Java Runtime Environment)
- Windows Command Line (module of Windows OS)

5. Proposed System Model

5.1 JSD (Jackson Structured Design)



5.2. Problem Analysis Chart

Data	Processing	Output	Solution Alternatives
account amount date address	login deposit withdraw	account balance account statement account enquiry	The statement for account (say x) for a particular period is going to be returned and printed out in the statement object.

6. Expected output

First, we have to register for RMI port

Let it be any random port 5169

Then login has to dine with right credentials

If successful:

Login for user gets valid for 5 minutes

Now ask for any option user wants among the provided functionalities

Let it be inquiry

If inquiry account number is typed

The current balance of account account number is amount

Now let the asked functionality be deposit

deposit account number: deposit amount

Successfully deposited deposit amount to account number

Note: All the underlined words are variable here that depends on the input from the user.

7. Implementation

7.1. Methodology

Client-Side Functions:

ATM.java

This is the client program, which connects to the bank using RMI and class methods of the remote bank object.

We interact with the Server via this Client program. The main function first connects to the server via RMI. We locate the server using the port number and the name lookup function. We call the Invalid Arguments function to check if we have given wrong input or not.

Next if the arguments are correct, we check which operation to execute. This is done by using the `getCommandLineArguments`. This function uses a switch case to check which operation is to be executed. Then it initializes the variables needed to be for further

execution. For example, if you specify the login operation, the username and password is initialized to the given values.

Now, that the given arguments have been initialized. The switch case prints out the required output for that specific operation. For example, if the operation is Login, the account number, session id, username, balance for the account you have just logged in for. If any invalid arguments or errors are faced we call the InvalidLogin or InvalidSession exception functions. These functions display the required error message and print the stack trace.

Error Functions:

InsufficientFundsException.java

When called this function displays the “Insufficient Funds” error message.

InvalidAccountException.java

When called this function displays the “Account with account number ----does not exist” error message.

InvalidArgumentException.java

When called this function displays the “Invalid command line arguments entered” error message.

InvalidLoginException.java

When called this function displays the “Your Login Details are Invalid” error message.

InvalidSessionException.java

When called this function displays the “Your session has timed out after 5 minutes of inactivity. Please Log In again” error message.

StatementException.java

When called this function displays the “Could not generate statement for given account and dates” error message.

Interface Functions:

BankInterface.java

This piece of code, inherits from the remote class and initializes the getAccountnum, getStartDate, getEndDate, getAccountName, getTransactions functions with their required parameters. It acts as a constructor for the statement class.

StatementInterface.java

This piece of code, inherits from the serializable class and initializes the login, deposit, withdraw, inquiry, getStatement, accountDetails functions with their required parameters. It acts as a constructor for the statement class.

Server-Side Functions:

Account.java

This contains the account class which holds user details, transactions and balance. It inherits from the Serializable class. It contains the Account, addTransactions, getUserName, setUserName, getBalance, setBalance, getPassword, setPassword, getAccountNumber, setAccountNumber, getTransactions and toString functions.

The variable nextAcNum sets the first account number from where every subsequent account number is incremented by one.

The Account functions holds the all the details of the account whose username and password are given as parameters.

All the functions use pointers to point to the current ongoing account.

Bank.java

This contains the Bank class which inherits from UnicastRemoteObject and Bank Interface. Here the three accounts have been initialized with usernames user1, user2, user3 and passwords pass1, pass2, pass3. The main function sets up the security manager for the server using the policies specified in the all.policy file. After setting up the security manager, the server is bound to the RMI port and the server is started. The login, deposit, withdraw, inquiry, getStatement, accountDetails, getAccount, checkSessionActive functions are defined here.

The login function check for the username and password. If it belongs to the list of accounts registered. It also creates and returns the sessionId for the particular account. If the login details are not valid it throws the InvalidLoginException.

The deposit function checks if the session is active. If so, it gets the particular account. Adds the amount to it using the setAmount function. Next the transaction object is created and a transaction is added referring to the amount and time and date details of the deposit. In the end it returns the updated account balance. If the account details are wrong, an InvalidAccountException is thrown.

The withdraw function checks if the session is active. If so, it gets the particular account details and deducts the withdraw amount from the account balance. Next the transaction object is created and a transaction is added referring to the amount and time and date details of the withdrawal. In the end it returns the updated account balance. If the account details are wrong, an InvalidAccountException is thrown. Else if the account doesn't have the sufficient funds, an InsufficientFundsException is thrown.

The inquiry function checks if the session is active. If so, it gets the particular account details and returns the particular account balance. If the account details are wrong, an `InvalidAccountException` is thrown.

The `getStatement` function checks if the session is active. If so, it gets the particular account. It also gets the dates from when to when we want the account statement. Next it creates a statement account object and calls the statement function to get the account statement. If the account details are wrong, an `InvalidAccountException` is thrown. Else if the account statement cannot be generated, an `StatementException` is thrown.

The `accountDetails` function gets the account details based on the `sessionID`. Each session has an associated account, so the accounts can be retrieved based on a session. It throws and `InvaiddSessionExpection` if the session isn't valid.

The `getAccount` function returns the particular account object for the given account number. If the account details are wrong, an `InvalidAccountException` is thrown.

The `checkSessionActive` function checks if the `sessionID` passed from client is in the sessions list and active. It prints session details and returns true if session is alive. If session is in list, but timed out, it is added to `deadSessions` list. This flags timed out sessions for `removeAll`. They will be removed using the `removeAll` function. It throws and `InvaiddSessionExpection` if the session isn't valid.

Session.java

This contains the session class which inherits `TimerTask`, a thread that can be called at certain time intervals. This allows the session to time to be incremented every second, and can be cancelled after 5mins (300s). It also inherits from the `Serializable` class. It contains the `Session`, `startTimer`, `run`, `isAlive`, `getClientId`, `getTimeAlive`, `getMaxSessionLenght`, `getAccount` and `toString` functions.

The `MAX_SESSION_LENGTH` variables is set to 300 (seconds) which is the time for a session to be alive.

The `Session` function creates a random `sessionId` for the given account. It sets the particular session to alive, creates and starts a new timer.

The `run` function increments the time the session has been alive and updates once every second, so it represents the number of seconds the session has been alive for. It keeps checking if the `Session` has completed its five minutes. If the `timeAlive` crosses 300, it sets the alive variable to false and stops the timer.

All the functions use pointers to point to the current ongoing account.

Statement.java

This contains the Statement class which inherits from the Serializable and Statement Interface classes. It contains the Statement, getAccountnum, getStartDate, getEndDate, getAccountName, getTransactions functions.

The Statement functions initializes the required variables.

The getTransactions function gets all the relevant transactions for the dates passed in statement function. It filters transactions based on the date provided. It then creates a list from these filtered transactions. This list is then returned by the function.

Transaction.java

It contains the Transaction class which tracks transaction dates, and other transaction details. It inherits from the Serializable class. It contains the Transaction, getDate, getAccountNumber, getType, setType, getAmount, setAmount and toString functions.

The Transaction function initializes the required variables.

The setAmount function updates the transaction amount and gets the new account balance, based on withdrawal or deposit.

The toString function formats the Date and Decimals to be output properly on the screen.

All the functions use pointers to point to the current ongoing account.

all.policy

This file grants permission using java.security.AllPermission to the security manager. Granting the server with full access.

Compilation Steps

Open a command prompt window. Navigate to the source code directory.

Type: javac -cp interfaces*.java

This will compile the interfaces into .class files.

Next Type: javac client\ATM.java

This will compile the client files.

Next Type: javac server*.java

This will compile the server files.

Next Type: rmiregistry 7777

This will start the RMI at port 7777.

In a new command window,

Type: java -Djava.security.policy=all.policy server.Bank 7777

This will start the server, grant it permissions and bind it to port 7777.

In a new command window,

Type: java client.ATM localhost 7777 login user1 pass1

This will start ATM client and login to user1.

7.2. Code

ATM.java

```
package client;

import exceptions.*;
import interfaces.BankInterface;
import server.Account;
import server.Statement;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.text.SimpleDateFormat;
import java.util.Date;

//Client program, which connects to the bank using RMI and class methods of the remote bank
object

public class ATM {

    static int serverAddress, serverPort, account;

    static String operation, username, password;

    static long sessionID, id=0;

    static double amount;

    static BankInterface bank;

    static Date startDate, endDate;
```

```

public static void main (String args[]) {
    try {
        //Parse the command line arguments into the program
        getCommandLineArguments(args);
        //Set up the rmi registry and get the remote bank object from it
        String name = "Bank";
        Registry registry = LocateRegistry.getRegistry(serverPort);
        bank = (BankInterface) registry.lookup(name);
        System.out.println("\n-----\nClient Connected" + "\n-----\n");
    } catch (InvalidArgumentException ie){
        ie.printStackTrace();
        System.out.println(ie);
    } catch (Exception e){
        e.printStackTrace();
        System.out.println(e);
    }
    double balance;

    //Switch based on the operation
    switch (operation){
        case "login":
            try {
                //Login with username and password
                id = bank.login(username, password);
                Account acc = bank.accountDetails(id);
                //Print account details
                System.out.println("-----\nAccount Details:\n-----
\n" +
                                "Account Number: " + acc.getAccountNumber() +

```

```

        "\nSessionID: " + id +
        "\nUsername: " + acc.getUserName() +
        "\nBalance: " + acc.getBalance() +
        "\n-----\n");

    System.out.println("Session active for 5 minutes");
    System.out.println("Use SessionID " + id + " for all other operations");

    //Catch exceptions that can be thrown from the server
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (InvalidLoginException e) {
        e.printStackTrace();
    } catch (InvalidSessionException e) {
        e.printStackTrace();
    }
    break;

case "deposit":
    try {
        //Make bank deposit and get updated balance
        balance = bank.deposit(account, amount, sessionID);
        System.out.println("Successfully deposited Rs. " + amount + " into account " +
account);

        System.out.println("New balance: Rs. " + balance);
        //Catch exceptions that can be thrown from the server
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (InvalidSessionException e) {
        System.out.println(e.getMessage());
    }
    break;

```

```

case "withdraw":

    try {

        //Make bank withdrawal and get updated balance

        balance = bank.withdraw(account, amount, sessionID);

        System.out.println("Successfully withdrew Rs. " + amount + " from account " +
account +

                                "\nRemaining Balance: Rs. " + balance);

        //Catch exceptions that can be thrown from the server
    } catch (RemoteException e) {

        e.printStackTrace();

    } catch (InvalidSessionException e) {

        System.out.println(e.getMessage());

    } catch (InsufficientFundsException e) {

        System.out.println(e.getMessage());

    }

    break;

case "inquiry":

    try {

        //Get account details from bank

        Account acc = bank.inquiry(account,sessionID);

        System.out.println("-----\nAccount Details:\n-----

\n" +

                                "Account Number: " + acc.getAccountNumber() +

                                "\nUsername: " + acc.getUserName() +

                                "\nBalance: Rs. " + acc.getBalance() +

                                "\n-----\n");

        //Catch exceptions that can be thrown from the server
    } catch (RemoteException e) {

        e.printStackTrace();

```

```

    } catch (InvalidSessionException e) {
        System.out.println(e.getMessage());
    }
    break;

case "statement":
    Statement s = null;
    try {
        //Get statement for required dates
        s = (Statement) bank.getStatement(account, startDate, endDate, sessionID);

        //format statement for printing to the window
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        System.out.print("-----\n");
        System.out.println("Statement for Account " + account + " between " +
            dateFormat.format(startDate) + " and " + dateFormat.format(endDate));
        System.out.print("-----\n");
        System.out.println("Date\t\t\tTransaction Type\tAmount\t\tBalance");
        System.out.print("-----\n");

        for(Object t : s.getTransations()) {
            System.out.println(t);
        }
        System.out.print("-----\n");
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (InvalidSessionException e) {
        System.out.println(e.getMessage());
    } catch (StatementException e) {

```



```
        System.out.println(e.getMessage());
    }
    break;

default:
    //Catch all case for operation that isn't one of the above
    System.out.println("Operation not supported");
    break;
}
}
```

```
public static void getCommandLineArguments(String args[]) throws
InvalidArgumentException{
    //Makes sure server, port and operation are entered as arguments
    if(args.length < 4) {
        throw new InvalidArgumentException();
    }

    //Parses arguments from command line
    //arguments are in different places based on operation, so switch needed here
    serverPort = Integer.parseInt(args[1]);
    operation = args[2];
    switch (operation){
        case "login":
            username = args[3];
            password = args[4];
            break;
        case "withdraw":
        case "deposit":
            amount = Double.parseDouble(args[4]);
```

```
        account = Integer.parseInt(args[3]);
        sessionID = Long.parseLong(args[5]);
        break;
    case "inquiry":
        account = Integer.parseInt(args[3]);
        sessionID = Long.parseLong(args[4]);
        break;
    case "statement":
        account = Integer.parseInt(args[3]);
        startDate = new Date(args[4]);
        endDate = new Date(args[5]);
        sessionID = Long.parseLong(args[6]);
        break;
    }
}
}
```

Bank.java

```
package server;

import exceptions.*;
import interfaces.*;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
//Bank Class which implements all the methods defined in the BankInterface
public class Bank extends UnicastRemoteObject implements BankInterface {

    private List<Account> accounts; // users accounts
    private List<Session> sessions, deadSessions;

    public Bank() throws RemoteException
    {
        super();
        //set up ArrayLists and create test accounts
        accounts = new ArrayList<>();
        sessions = new ArrayList<>();
        deadSessions = new ArrayList<>();

        accounts.add(new Account("user1", "pass1"));
        accounts.add(new Account("user2", "pass2"));
        accounts.add(new Account("user3", "pass3"));
    }

    public static void main(String args[]) throws Exception {
        try {
            //Set up securitymanager for server, and specify path to the policy file
            System.setProperty("java.security.policy", "C:\\Users\\Kanav\\Desktop\\PDC
PROJECT\\src\\all.policy");

            System.setSecurityManager(new SecurityManager());

            System.out.println("\n-----\nSecurity Manager Set");

            //Add bank to the RMI registry so it can be located by the client
            String name = "Bank";
```

```

BankInterface bank = new Bank();

Registry registry = LocateRegistry.getRegistry(Integer.parseInt(args[0]));
registry.rebind(name, bank);

System.out.println("Bank Server Bound");

System.out.println("Server Stared\n-----\n");
} catch (Exception e) {
    e.printStackTrace();
}
}

```

@Override

```

public long login(String username, String password) throws RemoteException,
InvalidLoginException {

    //Loop through the accounts to find the correct one for given username and password
    for(Account acc : accounts) {

        if(username.equals(acc.getUserName()) && password.equals(acc.getPassword())){

            System.out.println(">> Account " + acc.getAccountNumber() + " logged in");

            //Create a new session on Successfull login, and return ID to the client

            Session s = new Session(acc);

            sessions.add(s);

            return s.sessionId;

        }

    }

    //Throw exception if login details are not valid

    throw new InvalidLoginException();

}

```

@Override

```

public double deposit(int accountnum, double amount, long sessionID) throws
RemoteException, InvalidSessionException {

    //Check if user session is active, based on sessionID passed by client

```

```

if(checkSessionActive(sessionID)) {
    Account account;
    try {
        //Get the correct account
        account = getAccount(accountnum);
        account.setBalance(account.getBalance() + amount);
        //Create transaction object for this transaction and add to the account
        Transaction t = new Transaction(account, "Deposit");
        t.setAmount(amount);
        account.addTransaction(t);

        System.out.println(">> Rs. " + amount + " deposited to account " + accountnum +
"\n");

        //return balance to client
        return account.getBalance();
    } catch (InvalidAccountException e) {
        e.printStackTrace();
    }
}
return 0;
}

```

@Override

public double withdraw(int accountnum, double amount, long sessionID) throws
RemoteException,

```

    InsufficientFundsException, InvalidSessionException {
        //Check if user session is active, based on sessionID passed by client
        if(checkSessionActive(sessionID)) {
            try {
                //Get correct user account based on accountnum

```

```

    Account account = getAccount(accountnum);

    //Check if withdrawal can be made, based on account balance
    if (account.getBalance() > 0 && account.getBalance() - amount >= 0) {
        account.setBalance(account.getBalance() - amount);

        //create new Transaction and add to account
        Transaction t = new Transaction(account, "Withdrawal");
        t.setAmount(amount);
        account.addTransaction(t);

        System.out.println(">> Rs. " + amount + " withdrawn from account " + accountnum
+ "\n");

        //return updated balance
        return account.getBalance();
    }
} catch (InvalidAccountException e) {
    e.printStackTrace();
}
}

//Throw exception if account doesn't have enough money to withdraw
throw new InsufficientFundsException();
}

@Override

public Account inquiry(int accountnum, long sessionID) throws RemoteException,
InvalidSessionException {

    //Check if session is active based on sessionID that is passed in
    if(checkSessionActive(sessionID)) {
        try {
            //Get account and return to the client

```

```

        Account account = getAccount(accountnum);

        System.out.println(">> Balance requested for account " + accountnum + "\n");

        return account;
    } catch (InvalidAccountException e) {
        e.printStackTrace();
    }
}

return null;
}

@Override

public Statement getStatement(int accountnum, Date from, Date to, long sessionID) throws
RemoteException,

    InvalidSessionException, StatementException {
    //Check if the session is active based on sessionID from client
    if(checkSessionActive(sessionID)) {
        try {
            //Get correct user account
            Account account = getAccount(accountnum);

            System.out.println(">> Statement requested for account " + accountnum +
                " between " + from.toString() + " " + to.toString() + "\n");

            //Create new statement using the account and the dates passed from the client
            Statement s = new Statement(account, from, to);

            return s;
        } catch (InvalidAccountException e) {
            e.printStackTrace();
        }
    }

    //throw exception if statement cannot be generated
    throw new StatementException("Could not generate statement for given account and
dates");
}

```

```

}

@Override

public Account accountDetails(long sessionID) throws InvalidSessionException {

    //Get account details based on the session ID

    //Each session has an associated account, so the accounts can be retrieved based on a
    session

    //Used on the client for looking up accounts
    for(Session s:sessions){

        if(s.getClientId() == sessionID){

            return s.getAccount();

        }

    }

    //Throw exception if session isn't valid

    throw new InvalidSessionException();

}

private Account getAccount(int acnum) throws InvalidAccountException{

    //Loop through the accounts to find one corresponding to account number passed from the
    client

    //and return it

    for(Account acc:accounts){

        if(acc.getAccountNumber() == acnum){

            return acc;

        }

    }

    //Throw exception if account does not exist

    throw new InvalidAccountException(acnum);

}

private boolean checkSessionActive(long sessID) throws InvalidSessionException{

    //Loop through the sessions

    for(Session s : sessions){

        //Checks if the sessionID passed from client is in the sessions list and active

```



```
        if(s.getClientId() == sessID && s.isAlive()) {  
            //Prints session details and returns true if session is alive  
  
            System.out.println(">> Session " + s.getClientId() + " running for " +  
s.getTimeAlive() + "s");  
  
            System.out.println(">> Time Remaining: " + (s.getMaxSessionLength() -  
s.getTimeAlive()) + "s");  
  
            return true;  
        }  
  
        //If session is in list, but timed out, add it to deadSessions list  
  
        //This flags timed out sessions for removeAll  
  
        //They will be removed next time this method is called  
  
        if(!s.isAlive()) {  
            System.out.println("\n>> Cleaning up timed out sessions");  
  
            System.out.println(">> SessionID: " + s.getClientId());  
  
            deadSessions.add(s);  
        }  
    }  
  
    System.out.println();  
  
    // cleanup dead sessions by removing them from sessions list  
    sessions.removeAll(deadSessions);  
  
    //throw exception if sessions passed to client is not valid  
    throw new InvalidSessionException();  
}  
}
```

7.3. Screenshots

```

C:\Command Prompt - rmiregistry 6969
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Kanav>cd "Desktop\PDC PROJECT\src"

C:\Users\Kanav\Desktop\PDC PROJECT\src>javac -cp interfaces\*.java

C:\Users\Kanav\Desktop\PDC PROJECT\src>jar cvf bank.jar interfaces\*.class
added manifest
adding: interfaces/BankInterface.class(in = 701) (out= 366)(deflated 47%)
adding: interfaces/StatementInterface.class(in = 344) (out= 234)(deflated 31%)

C:\Users\Kanav\Desktop\PDC PROJECT\src>javac client\ATM.java
Note: client\ATM.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Kanav\Desktop\PDC PROJECT\src>javac server\*.java

C:\Users\Kanav\Desktop\PDC PROJECT\src>rmiregistry 6969
■

C:\Command Prompt - java -Djava.security.policy=all.policy.server.Bank 6969

C:\Users\Kanav\Desktop\PDC PROJECT\src>java -Djava.security.policy=all.policy.server.Bank 6969

-----
Security Manager Set
Bank Server Bound
Server Stared
-----

>> Account 88769912 logged in
>> Session 443067 created

>> Session 443067 running for 55s
>> Time Remaining: 245s
>> Balance requested for account 88769912

>> Session 443067 running for 71s
>> Time Remaining: 229s
>> Rs. 5000.0 deposited to account 88769912

>> Session 443067 running for 86s
>> Time Remaining: 214s
>> Rs. 1.0 withdrawn from account 88769912

>> Session 443067 running for 102s
>> Time Remaining: 198s
>> Balance requested for account 88769912


>> Session 443067 running for 141s
>> Time Remaining: 159s
>> Statement requested for account 88769912 between Fri Oct 07 00:00:00 IST 2016 Fri Nov 10 00:00:00 IST 2017

>> Session 443067 running for 162s
>> Time Remaining: 138s
>> Statement requested for account 88769912 between Fri Oct 07 00:00:00 IST 2016 Fri Nov 10 00:00:00 IST 2017

>> Account 88769913 logged in
>> Session 183850 created

>> Account 88769913 logged in
>> Session 641941 created

```

 Command Prompt

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 login user1 pass1

Client Connected

Account Details:

Account Number: 88769912
SessionID: 443067
Username: user1
Balance: 0.0

Session active for 5 minutes
Use SessionID 443067 for all other operations

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 inquiry 88769912 443067

Client Connected

Account Details:


Account Number: 88769912
Username: user1
Balance: Rs. 0.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 deposit 88769912 5000 443067

Client Connected

Successfully deposited Rs. 5000.0 into account 88769912
New balance: Rs. 5000.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 withdraw 88769912 1 443067

 Command Prompt

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 deposit 88769912 5000 443067

Client Connected

Successfully deposited Rs. 5000.0 into account 88769912
New balance: Rs. 5000.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 withdraw 88769912 1 443067

Client Connected

Successfully withdrew Rs. 1.0 from account 88769912
Remaining Balance: Rs. 4999.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 inquiry 88769912 443067

Client Connected

Account Details:

Account Number: 88769912
Username: user1
Balance: Rs. 4999.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 statement 88769912 10/07/2016 11/10/17 443067

Client Connected

Statement for Account 88769912 between 07/10/2016 and 10/11/2017

Date	Transaction Type	Amount	Balance
------	------------------	--------	---------

Command Prompt

Session active for 5 minutes
Use SessionID 234859 for all other operations

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 inquiry 88769912 234859

Client Connected

Account Details:

Account Number: 88769912
Username: user1
Balance: Rs. 4999.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 deposit 88769912 1 234859

Client Connected

Successfully deposited Rs. 1.0 into account 88769912
New balance: Rs. 5000.0

C:\Users\Kanav\Desktop\PDC PROJECT\src>java client.ATM localhost 6969 statement 88769912 10/07/2016 11/10/2017 234859

Client Connected

Statement for Account 88769912 between 07/10/2016 and 10/11/2017

Date	Transaction Type	Amount	Balance
06/11/2017 10:53:02	Deposit	5000.0	5,000
06/11/2017 10:53:17	Withdrawal	1.0	4,999
06/11/2017 11:38:53	Deposit	1.0	5,000

8. Conclusion

In this day and age of developing advancements, endeavors are moving towards the Internet for trade and business. Individuals are surging towards the web based business applications for their everyday needs, which thus are making the Internet extremely well known. Internet Banking has given both an open door and a test to conventional managing an account. In the quickly developing world, managing an account is a need, which takes a great deal of time from our bustling schedule. How Distributed applications like web based Banking, can be created without much of a stretch utilizing Java and its conveyed model of distributed architecture, is shown in this paper. This project as intended has helped us to learn a lot how to write distributed programs in Java. We created a program with the aim of implementing a client-server based program which communicated via Java RMI. Here the client i.e. terminal initiates an operation by calling a remote method on the bank server to execute some basic transaction functionalities but first of all login is

required to authenticate the user account for which simple array based data structure has been used.

9. Future Development Possibilities

This project can be enhanced by incorporating various other functionalities or features like enhanced security with encrypted implementation of login functionality. Also, GUI implementation instead of the current command line implementation will be a more practical and aesthetically pleasing method to go about it.

10. Reference

- [1] DISTRIBUTED SYSTEMS: Principles and paradigms by Andrew S. Tanenbaum and Maarten Van Steen
- [2] Distributed Online Banking by Mahmood Akhtar and Kamyar Dezhgosha
- [3] Maassen, Jason, et al. "Efficient Java RMI for parallel programming." ACM Transactions on Programming Languages and Systems 23.6 (2001): 747-775.
- [4] Distributed System and its Role in HealthCare System Anjali Saini¹ , P.K.Yadav² Department of Computer Science, Singhania University, Rajasthan, India
- [5] Buchmann, Alejandro P., et al. "A transaction model for active distributed object systems." (1992): 123-158.
- [6] Raman, Arun, et al. "Speculative parallelization using software multi-threaded transactions." ACM SIGARCH computer architecture news. Vol. 38. No. 1. ACM, 2010.
- [7] Irani, Keki B., and Hing-Lung Lin. "Queueing network models for concurrent transaction processing in a database system." Proceedings of the 1979 ACM SIGMOD international conference on Management of data. ACM, 1979.
- [8] Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control." ACM Transactions on Database Systems (TODS) 6.2 (1981): 213-226.
- [9] Keyser Jr, George T., Carl R. Fosler, and Jeffrey D. Johnson. "Portable personal-banking system." U.S. Patent No. 5,025,373. 18 Jun. 1991.

Plagiarism Check:

Completed: 100% Checked	0% Plagiarism	100% Unique
100% Checked		
In this implementation of the distributed banking system we use java to create a modular executable d...	- Unique	
This is reminiscent to virtual client-server system as the data structure will serve similar functionality a...	- Unique	
The client will first login (necessary for server to authorize the client) and after doing so client will have...	- Unique	
The pros for decentralizing the processing conquer the cons so, here we are implementing a distribut...	- Unique	
The following is the report for my project done at VIT University in Fall semester 2017-2018 session.	- Unique	
The project has been done in a group of three with a great assist of our faculty assigned to the course.	- Unique	
In this day and age of developing advancements, endeavors are moving towards the Internet for trade...	- Unique	
Internet Banking has given both an open door and a test to conventional managing an account.	- Unique	
Setting off to a branch or ATM or paying bills by paper look at and mailing them, and adjusting checkb...	- Unique	
For all banks, internet saving money is a capable apparatus to increase new clients while it serves to ...	- Unique	
Current and sudden increase in services moving to on demand anywhere access platform can be cre...	- Unique	
One such service is the banking sector, as every bank is now extensively pushing towards the goal of ...	- Unique	
if this is not accomplish the system will be obsolete. Another major concern is making the system robu...	- Unique	
In this whole process of developing prototype of distributed banking system, we have followed a effect...	- Unique	
So, with this trend in mind we tried to find an application of the course CSE-4001 Parallel and Distribu...	- Unique	
the independent clients/users which will try to simultaneously access a server emulated by a databas...	- Unique	

Completed: 100% Checked	0% Plagiarism	100% Unique
100% Checked		
Paper [1] sheds light on the basic introduction of the concepts of Distributed system most noticeably t...	- Unique	
"A distributed system is a collection of independent computers that appears to its users as a single co...	- Unique	
forms a vital and defining characteristic of distributed system i.e. this makes these otherwise autonom...	- Unique	
Further, an important characteristic of a distributed system is that the way the autonomous units differ...	- Unique	
These characteristics provide distributed system ability to be more scalable and robust. In principle, al...	- Unique	
A sent-out system will normally be consistently available, although perhaps some parts may be briefly...	- Unique	
Because you'll be able to build sent out systems will not necessarily mean that it's a good notion.	- Unique	
In such a section we discuss four important goals that needs to be achieved to make creating a distrib...	- Unique	
Distributed systems contain autonomous pcs that interact to give the looks of an individual coherent s...	- Unique	
Another benefit is that whenever properly designed, allocated systems level well with regards to the si...	- Unique	
Nevertheless, there may be substantial interest worldwide in building and putting in distributed systems.	- Unique	
However, this syndication transparency not only comes at a performance price, however in sensible si...	- Unique	
Later, when assumptions are decreased, it may grow to be difficult to cover up unwanted behavior. An...	- Unique	
Other pitfalls include let's assume that the network is reliable, static, secure, and homogeneous.	- Unique	
In this paper [2], the author reflects on the changing trend of today's consumer, demanding easier and...	- Unique	
Particularly focusing on banking system this paper states that due the introduction of online banking s...	- Unique	
This paper [2] shows, how Distributed applications like online Bank, may easily be developed using B...	- Unique	
J2EE, as allocated component structures, is the right solution in expanding web applications.	- Unique	

Completed: 100% Checked	0% Plagiarism	100% Unique
100% Checked		
Next step was the communication, which was accomplished by Java RMI for which we refer to paper [...]	- Unique	
The paper [3] further talks about the different methods of using the Java RMI and how to implement it [...]	- Unique	
Firstly, the Manta Java System and the compares it to the Sun RMI implementation. Further the talk a...	- Unique	
The null-RMI latency of Manta RMI over Myrinet is 37 us, only 6 us slower when compared to a C-bas...	- Unique	
Next, we provide a detailed performance assessment between your Manta and Sunshine RMI protoco...	- Unique	
The results show that the Manta standard protocol ends up with 1.8 to 3.4 times higher speedups for f...	- Unique	
In Section 3, we provide a detailed examination of the communication performance of our bodies. In S...	- Unique	
The earlier traditional banking systems had only single tier systems, supported centralized data bases...	- Unique	
Traditional systems provided only read access to the users.	- Unique	
In this paper [4], author shown a research point of view on sent out system in medical care sector.	- Unique	
Also today's data recording types of procedures in almost all of health centers leave much to be desir...	- Unique	
Within the last couple of years of the nineteenth hundred years and early couple of years of the twenti...	- Unique	
Nursing homes also implemented it by transmitting electrocardiograms over mobile phone lines. We w...	- Unique	
Due to the demerits of the DOM model a need for a new more powerful transaction model arose.	- Unique	
Correctness criteria is the degree of concurrency the model has. This separation of transaction model ...	- Unique	
heterogeneous data source systems [Litwin, 1988; Gupta, 1989; Sheth and Larson, 1990].	- Unique	
The DOM (Distributed Subject Management) job at GTE Laboratories addresses specifically these iss...	- Unique	
included [Manola, 1988 and 1989; Manola and Buchmann, 1990].	- Unique	

Completed: 100% Checked	19% Plagiarism	81% Unique
100% Checked		
Right concept implementation has the ability to exponentially elevating the performance measures of ...	- Unique	
But most software and hardware thread level speculation memory system are not sufficient as they ar...	- Plagiarized	
With the right techniques, multicore architectures might be able to continue the exponential performan...	- Unique	
Recently-proposed code parallelization techniques, such as those by Bridges et al.	- Plagiarized	
However, most software and hardware Thread-Level Speculation (TLS) ram systems and transaction...	- Plagiarized	
This paper proposes a Software MTX (SMTX) system that catches the applicability and performance ...	- Plagiarized	
Primary aim of development of these models as discussed in paper [7] is to investigate the effects of t...	- Unique	
These models also closely examine cause-effect relationships low cost concurrent transaction proces...	- Unique	
Several undesired situations might occur if the distributed usage of the data source by concurrent ven...	- Unique	
In order to avoid occurrence of the undesirable situations also to preserve the persistence of the data ...	- Unique	
the execution of the same transactions in a few serial order. The currently common concurrency contr...	- Unique	
Also, transaction backup as a control mechanism is vital point. Consider the challenge of providing dis...	- Unique	
Any collection of accesses to the repository that preserves the integrity constraints of the info is name...	- Unique	
(1) The quantity of data is sufficiently great that at any moment only a small fraction of the repository c...	- Unique	
In both situations the hardware will be underutilized if the amount of concurrency is too low.	- Unique	
The terminals have minimal keyboard and display functionality. The control and mode of operations re...	- Unique	
A lightweight personal bank operating system comprises a bunch computer with least one lightweight ...	- Unique	
The non-public terminal includes a computer keyboard and display capabilities with mininal key requir...	- Unique	
Subsequently, future changes or improvements to bank services may be executed without the change...	- Unique	
Yet, an individual terminal offering system works with with a offering system including office or home p...	- Unique	
The non-public terminal real estate may consist of a prominent surface for data access and screen an...	- Unique	

Completed: 100% Checked	0% Plagiarism	100% Unique
100% Checked		
All these papers are centered on the basic distributed banking system implantation analysis. It goes w...	- Unique	
Also, portable banking system has been studied on the basis of the previous analysis. The aim of this ...	- Unique	
user account for which simple array based data structure has been used.	- Unique	
Once they are granted the access they can perform basic transaction operations like deposit, withdra...	- Unique	
- deposit: in this operation the user account's balance would get increased by the amount entered.	- Unique	
- inquiry: using this operation the user would get to know his/her balance of respective account. - getS...	- Unique	
terminal initiates an operation by calling a remote method on the bank server to execute one of the ab...	- Unique	
The session ID acts as an authentication token that must be passed for each of the other remote met...	- Unique	
First, we have to register for RMI port Then login has to dine with right credentials	- Unique	
The current balance of account account number is amount Note: All the underlined words are variable...	- Unique	
The main function first connects to the server via RMI. We locate the server using the port number an...	- Unique	
This function uses a switch case to check which operation is to be executed. Then it initializes the vari...	- Unique	
The switch case prints out the required output for that specific operation.	- Unique	
If any invalid arguments or errors are faced we call the InvalidLogin or InvalidSession exception functi...	- Unique	
When called this function displays the "Invalid command line arguments entered" error message. Whe...	- Unique	
When called this function displays the "Could not generate statement for given account and dates" err...	- Unique	
It acts as a constructor for the statement class.	- Unique	
It acts as a constructor for the statement class.	- Unique	

Checking...	0% Plagiarism	100% Unique
100% Checked		
This contains the account class which holds user details, transactions and balance. It contains the Ac...	- Unique	
The Account functions holds the all the details of the account whose username and password are giv...	- Unique	
Here the three accounts have been initialized with usernames user1, user2, user3 and passwords pas...	- Unique	
After setting up the security manager, the server is bound to the RMI port and the server is started.	- Unique	
If it belongs to the list of accounts registered. It also creates and returns the sessionId for the particu...	- Unique	
Adds the amount to it using the setAmount function.	- Unique	
In the end it returns the updated account balance. If the account details are wrong, an InvalidAccount...	- Unique	
Next the transaction object is created and a transaction is added referring to the amount and time and...	- Unique	
Else if the account doesn't have the sufficient funds, an InsufficientFundsException is thrown. The inq...	- Unique	
The getStatement function checks if the session is active. It also gets the dates from when to when w...	- Unique	
Else if the account statement cannot be generated, an StatementException is thrown. The accountDet...	- Unique	
The getAccount function returns the particular account object for the given account number. If the acc...	- Unique	
If session is in list, but timed out, it is added to deadSessions list. They will be removed using the rem...	- Unique	
This allows the session to time to be incremented every second, and can be cancelled after 5mins (30...	- Unique	
The MAX_SESSION_LENGTH variables is set to 300 (seconds) which is the time for a session to be ...	- Unique	
The run function increments the time the session has been alive and updates once every second, so i...	- Unique	
All the functions use pointers to point to the current ongoing account. This contains the Statement cla...	- Unique	
It filters transactions based on the date provided. It then creates a list from these filtered transactions.	- Unique	
It contains the Transaction, getDate, getAccountNumber, getType, setType, getAmount, setAmount a...	- Unique	
The toString function formats the Date and Decimals to be output properly on the screen. All the functi...	- Unique	

Completed: 100% Checked	0% Plagiarism	100% Unique
100% Checked		
In this day and age of developing advancements, endeavors are moving towards the Internet for trade...		- Unique
Internet Banking has given both an open door and a test to conventional managing an account.		- Unique
How Distributed applications like web based Banking, can be created without much of a stretch utilizin...		- Unique
We created a program with the aim of implementing a client-server based program which communicat...		- Unique
user account for which simple array based data structure has been used.		- Unique
Also, GUI implementation instead of the current command line implementation will be a more practical...		- Unique

We have tried to get it checked by Kannan sir, but we haven't got a reply. So we have gone for online plagiarism check.

The screenshot shows a Gmail interface with a 'Plagiarism Check Request' email from Shivang Raj (shivang.raj2015@vit.ac.in) to Kannan V. The email body reads: 'I kindly request you to check the plagiarism for the following document and kindly mail me the report at the earliest so that I can make changes in my document. Thanking You'. A thumbnail of a document titled 'Internet Banking System' is visible. The email is dated Nov 7 (1 day ago). The bottom of the screen shows a Windows taskbar with various application icons and the system clock displaying 22:02 on 08-11-2017.