RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSE301A
Course Name	Distributed Systems
Programme	B. Tech. in CSE
Department	Computer Science and Engineering
Faculty	FET

Name of the Student	ARYAN JALALI
Reg. No	17ETCS002033
Semester/Year	06/2020
Course Leader/s	Chaitra S

:

Declaration Sheet				
Student Name	ARYAN JALALI			
Reg. No	17ETCS002033			
Programme	B. Tech. in CSE		Semester/Year	06/2020
Course Code	CSE301A			
Course Title	Distributed Systems			
Course Date		to		
Course Leader	Chaitra S			

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	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Faculty of Engineering & Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech. in Computer Science and Engineering
Semester/Batch	6th /2017		
Course Code	CSE301A	Course Title	Distributed Systems
Course Leader	Chaitra S		

Assignment

Register No.		Name of Student	

Sections		Marking Scheme	Max Marks	First Examiner Marks	Second Examiner Marks
Part-A	A1.1	Design of banking application	05		
	A1.2	Implementation of banking application	05		
	A1.3	Testing of the developed banking application	02		
	A1.4	Demonstration	05		
		Part-A Max Marks	**17**		
Part-B	B1.1	Importance of RPC	03		
	B1.2	Importance of RMI	03		
	B1.3	Differences between RPC and RMI	02		
		Part-B Max Marks	**08**		
		Total Assignment Marks	**25**		

Course Marks Tabulation

Component- 1(B) Assignment	First Examiner	Remarks	Second Examiner	Remarks
A				
B				
Marks (out of 25)				

Signature of First Examiner Signature of Second Examiner

Please note:

1. Documental evidence for all the components/parts of the assessment such as the reports, photographs, laboratory exam / tool tests are required to be attached to the assignment report in a proper order.
2. The First Examiner is required to mark the comments in RED ink and the Second Examiner's comments should be in GREEN ink.
3. The marks for all the questions of the assignment have to be written only in the **Component – CET B: Assignment** table.
4. If the variation between the marks awarded by the first examiner and the second examiner lies within +/- 3 marks, then the marks allotted by the first examiner is considered to be final. If the variation is more than +/- 3 marks then both the examiners should resolve the issue in consultation with the Chairman BoE.

Solution to Question No. A1:

A1.1 Design of an effective solution to the above scenario and description of techniques used

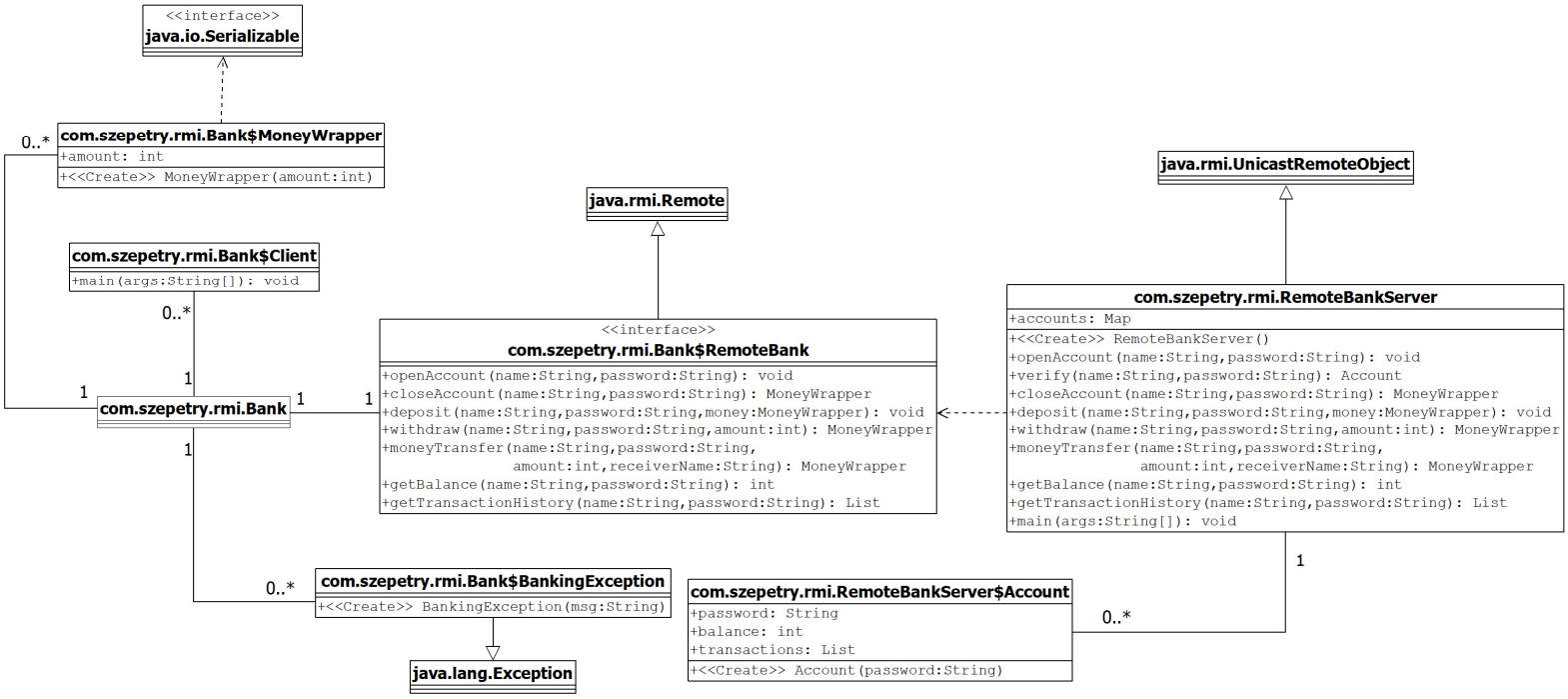


Figure A1.1: Design for the Banking application using RMI depicted via Low Level Class Diagram

My solution to the given scenario is using Remote method invocation (RMI) for handling single server, multiple clients. In this model, the server defines objects that clients can use remotely. Clients invoke methods of a remote object exactly as if it were a local object running in the same virtual machine as the client. RMI hides the underlying mechanism for transporting method arguments and return values across the network.

RMI is available as a package in Java which provides built-in methods needed for this scenario.

RMI allows synchronization of methods which allows mutual exclusion between methods of different clients. This makes sure that the other invocations made by other clients don't affect the busy client.

The Bank class contains an interface (Can be invoked remotely) for the client/server as well as the client class itself. It also contains a wrapper class which stores transaction details as serializable. Client class inside this bank class drives the main class for Bank class. It defines the arguments to be passed to the server on execution and which registry to look for to find the server.

The RemoteBankServer class is a remote object which implements the interface RemoteBank from the Bank class. This is done to make sure all the methods defined in RemoteBank are defined in the server class. RemoteBankServer class maintains all the accounts as a hashmap. It defines all the methods required by the client such as deposit, withdraw, getBalance, history, money transfer, etc. All the methods are

synchronized which makes sure that mutual exclusion is promised. The main class creates a registry for the server to start on. This allows the client to make a connection with the server.

This mutual exclusion will make sure that no conflict pairs arise.

A1.2 Implementation of banking application to reflect correct and consistent states of account balance when multiple clients initiate transactions.

new 1.txt | Bank.java | RemoteBankServer.java

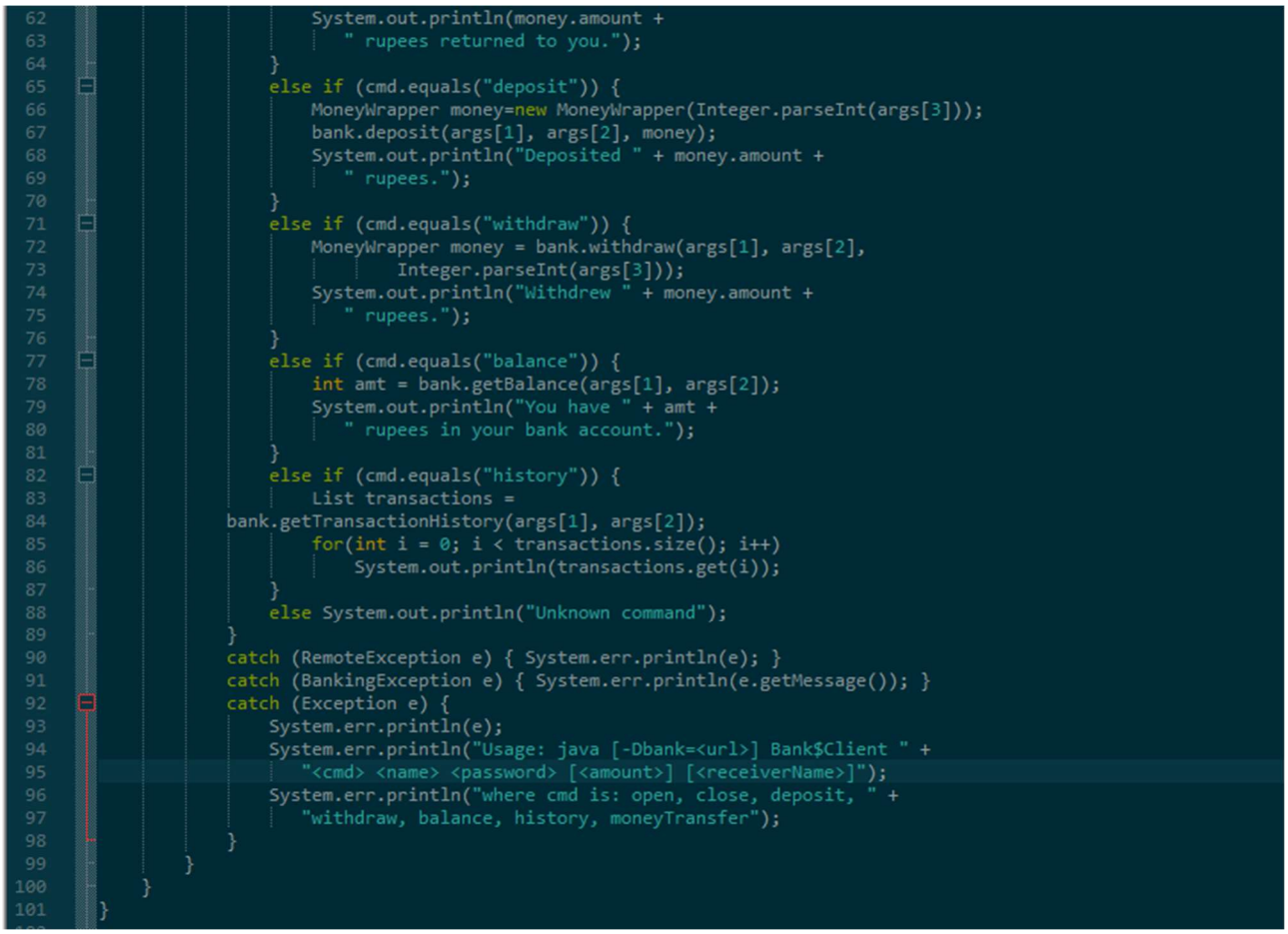
```
1  package com.szepetry.rmi;
2  import java.rmi.*;
3  import java.rmi.server.*;
4  import java.util.*;
5  import com.szepetry.rmi.Bank.*;
6  import java.rmi.registry.LocateRegistry;
7  import java.rmi.registry.Registry;
8
9  public class RemoteBankServer extends UnicastRemoteObject implements RemoteBank
10 {
11     class Account {
12         String password;
13         int balance;
14         List transactions = new ArrayList();
15         Account(String password) {
16             this.password = password;
17             transactions.add("Account opened at " + new Date());
18         }
19     }
20
21     Map accounts = new HashMap();
22
23     public RemoteBankServer() throws RemoteException { super(); }
24
25     public synchronized void openAccount(String name, String password)
26     throws RemoteException, BankingException
27     {
28         if (accounts.get(name) != null)
29             throw new BankingException("Account already exists.");
30         Account acct = new Account(password);
31         accounts.put(name, acct);
32     }
33
34     Account verify(String name, String password) throws BankingException {
35         synchronized(accounts) {
36             Account acct = (Account)accounts.get(name);
37             if (acct == null) throw new BankingException("No such account");
38             if (!password.equals(acct.password))
39                 throw new BankingException("Invalid password");
40             return acct;
41         }
42     }
43
44     public synchronized MoneyWrapper closeAccount(String name, String password)
45     throws RemoteException, BankingException
46     {
47         Account acct;
48         acct = verify(name, password);
49         accounts.remove(name);
50         synchronized (acct) {
51             int balance = acct.balance;
52             acct.balance = 0;
53             return new MoneyWrapper(balance);
54         }
55     }
56
57     public void deposit(String name, String password, MoneyWrapper money)
58     throws RemoteException, BankingException
59     {
60         Account acct = verify(name, password);
61         synchronized(acct) {
```

```
62              acct.balance += money.amount;
63              acct.transactions.add("Deposited " + money.amount +
64                      " on " + new Date());
65          }
66      }
67
68      public MoneyWrapper withdraw(String name, String password, int amount)
69      throws RemoteException, BankingException
70      {
71          Account acct = verify(name, password);
72          synchronized(acct) {
73              if (acct.balance < amount)
74                  throw new BankingException("Insufficient Funds");
75              acct.balance -= amount;
76              acct.transactions.add("Withdrew " + amount + " on "+new Date());
77              return new MoneyWrapper(amount);
78          }
79      }
80
81      public MoneyWrapper moneyTransfer(String name, String password,
82              int amount, String receiverName)
83      throws RemoteException, BankingException
84      {
85          Account acct = verify(name, password);
86          Account acctReceiver=(Account)accounts.get(receiverName);
87          if (acctReceiver == null) throw new BankingException("No such account");
88          synchronized(acct) {
89              if (acct.balance < amount)
90                  throw new BankingException("Insufficient Funds");
91              acct.balance -= amount;
92              acctReceiver.balance += amount;
93              acct.transactions.add("Sent " + amount +" to Account "+
94                      receiverName+" on "+new Date());
95              acctReceiver.transactions.add("Received " +amount+" from Account "+
96                      name+" on "+new Date());
97              return new MoneyWrapper(amount);
98  //          }
99      }
100     }
101
102     public int getBalance(String name, String password)
103     throws RemoteException, BankingException
104     {
105         Account acct = verify(name, password);
106         synchronized(acct) { return acct.balance; }
107     }
108
109     public List getTransactionHistory(String name, String password)
110     throws RemoteException, BankingException
111     {
112         Account acct = verify(name, password);
113         synchronized(acct) { return acct.transactions; }
114     }
115
116
117     public static void main(String[] args) {
118         try {
119             Registry reg = LocateRegistry.createRegistry(1099);
120             RemoteBankServer bank = new RemoteBankServer();
121             String name = System.getProperty("bankname", "MainBankServer");
122             reg.rebind(name, bank);
123             System.out.println(name + " is now running. "
124                     + "Please connect using Client.\nUsage: java -jar"
125                     + " \"distributedSysTestProj5.jar\" [options]");
126             System.out.println("[Options]:\nOPEN ID PASS\nCLOSE ID PASS\n"
127                     + "WITHDRAW ID PASS amt\nDEPOSIT ID PASS amt\n"
128                     + "HISTORY ID PASS\nBALANCE ID PASS\n"
129                     + "TRANSFER ID PASS amt receivername\n");
130         }
131         catch (Exception e) {
132             System.err.println(e);
133             System.err.println("Usage: java [-Dbankname=<name>] " +
134                     "com.szepetry.rmi.RemoteBankServer");
135             System.exit(1);
136         }
137     }
138 }
```

Figure A1.2: RemoteBankServer.java: The implementation of the server

new 1.txt | Bank.java | RemoteBankServer.java

```java
package com.szepetry.rmi;

import java.rmi.*;
import java.util.List;

public class Bank {

    public interface RemoteBank extends Remote {
        public void openAccount(String name, String password)
        throws RemoteException, BankingException;

        public MoneyWrapper closeAccount(String name, String password)
        throws RemoteException, BankingException;

        public void deposit(String name, String password, MoneyWrapper money)
        throws RemoteException, BankingException;

        public MoneyWrapper withdraw(String name, String password, int amount)
        throws RemoteException, BankingException;

        public int getBalance(String name, String password)
        throws RemoteException, BankingException;

        public List getTransactionHistory(String name, String password)
        throws RemoteException, BankingException;

        public MoneyWrapper moneyTransfer(String name, String password,
                int amount,String receiverName) //-----------------
            throws RemoteException, BankingException;
    }


    public static class MoneyWrapper implements java.io.Serializable {
        public int amount;
        public MoneyWrapper(int amount) { this.amount = amount; }
    }

    public static class BankingException extends Exception {
        public BankingException(String msg) { super(msg); }
    }


    public static class Client {
        public static void main(String[] args) {
            try {
                String url = System.getProperty("bank", "rmi:///MainBankServer");
                RemoteBank bank = (RemoteBank) Naming.lookup(url);
                String cmd = args[0].toLowerCase();

                if (cmd.equals("open")) {
                    bank.openAccount(args[1], args[2]);
                    System.out.println("Account opened.");
                }
                else if (cmd.equals("transfer")) {
                    MoneyWrapper money = bank.moneyTransfer(args[1], args[2],
                            Integer.parseInt(args[3]), args[4]);
                    System.out.println(money.amount +
                       " rupees transfered to "+args[4]+".");
                }
                else if (cmd.equals("close")) {
                    MoneyWrapper money = bank.closeAccount(args[1], args[2]);
```



```
                    System.out.println(money.amount +
                        " rupees returned to you.");
                }
                else if (cmd.equals("deposit")) {
                    MoneyWrapper money=new MoneyWrapper(Integer.parseInt(args[3]));
                    bank.deposit(args[1], args[2], money);
                    System.out.println("Deposited " + money.amount +
                        " rupees.");
                }
                else if (cmd.equals("withdraw")) {
                    MoneyWrapper money = bank.withdraw(args[1], args[2],
                            Integer.parseInt(args[3]));
                    System.out.println("Withdrew " + money.amount +
                        " rupees.");
                }
                else if (cmd.equals("balance")) {
                    int amt = bank.getBalance(args[1], args[2]);
                    System.out.println("You have " + amt +
                        " rupees in your bank account.");
                }
                else if (cmd.equals("history")) {
                    List transactions =
            bank.getTransactionHistory(args[1], args[2]);
                    for(int i = 0; i < transactions.size(); i++)
                        System.out.println(transactions.get(i));
                }
                else System.out.println("Unknown command");
            }
            catch (RemoteException e) { System.err.println(e); }
            catch (BankingException e) { System.err.println(e.getMessage()); }
            catch (Exception e) {
                System.err.println(e);
                System.err.println("Usage: java [-Dbank=<url>] Bank$Client " +
                    "<cmd> <name> <password> [<amount>] [<receiverName>]");
                System.err.println("where cmd is: open, close, deposit, " +
                    "withdraw, balance, history, moneyTransfer");
            }
        }
    }
}
```

Figure A1.3: Bank.java: The implementation of the client

Discussion for the server side implementation shown in Figure A1.2:

The RemoteBankServer class is a remote object which was done by extending it with UnicastRemoteObject from java.rmi.server. This class also implements the interface RemoteBank in the Bank class.

RemoteBankServer class maintains all the accounts as a hashmap.

It defines all the methods required by the client such as openAccount, closeAccount, deposit, withdraw, getBalance, getTransactionHistory moneyTransfer.

All the methods are synchronized which makes sure that mutual exclusion is promised. This mutual exclusion will make sure that no conflict pairs arise. Java RMI Synchronization handles conflicts as it executes the methods in separate thread (locking them when in use and unlocking when not in) making them run in a safe manner.

The main class creates a registry for the server to start on. This allows the client to make a connection with the server.

Discussion for the server side implementation shown in Figure A1.3:

The Bank class contains an interface called RemoteBank which creates the structure of the methods needed in any class object that implements on this interface. Bank class also contains a wrapper class which stores transaction details as serializable.

There is also an exception class defined which is called when the try block fails and an exception is caught. Client class inside this bank class drives the main class for Bank class. It defines the arguments to be passed to the server on execution and which registry to look for to find the server.

A1.3 Testing of the developed banking application

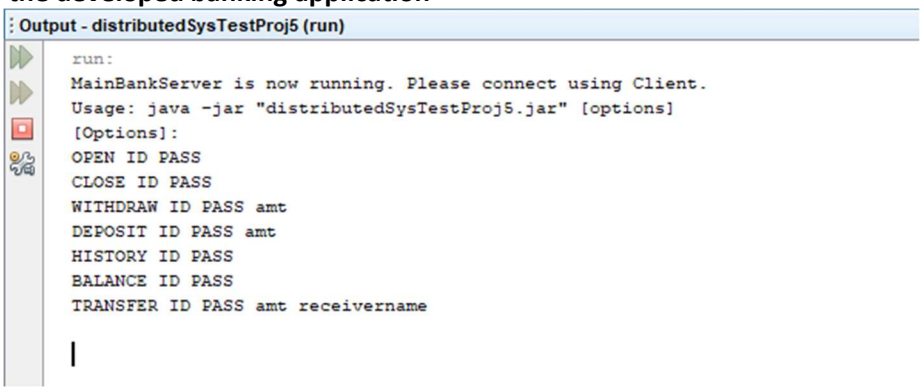


Figure A1.4: Running RemoteBankServer.java file

Performing non conflict operations with the client to show functionality of the Client options.

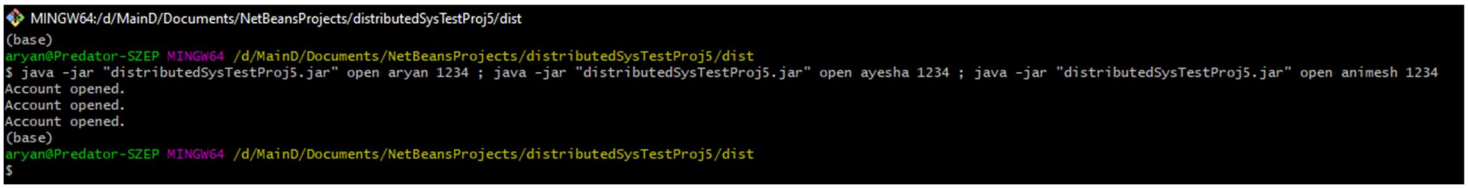


Figure A1.5: Opening three accounts using Bank.java 's jar file and giving parameters for opening account.

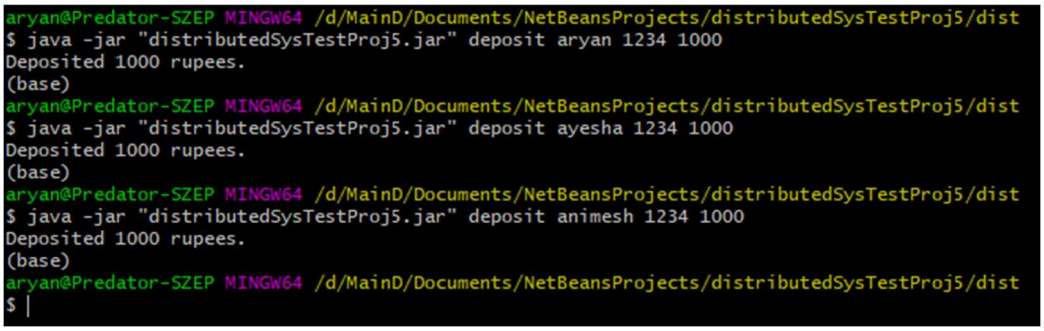


Figure A1.6: Using deposit parameter while running to add Rs. 1000 to all accounts.

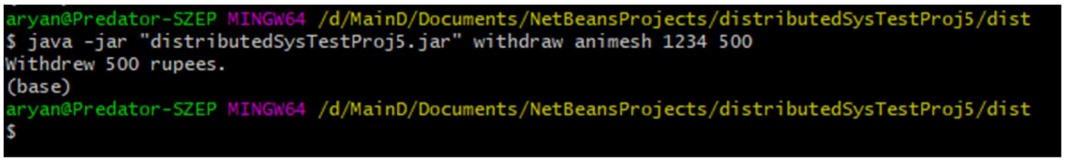


Figure A1.7: Using withdraw parameter on account animesh.

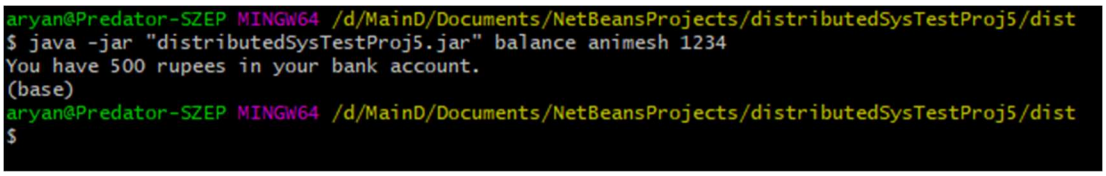


Figure A1.8: Using balance parameter on account animesh to show change in balance.

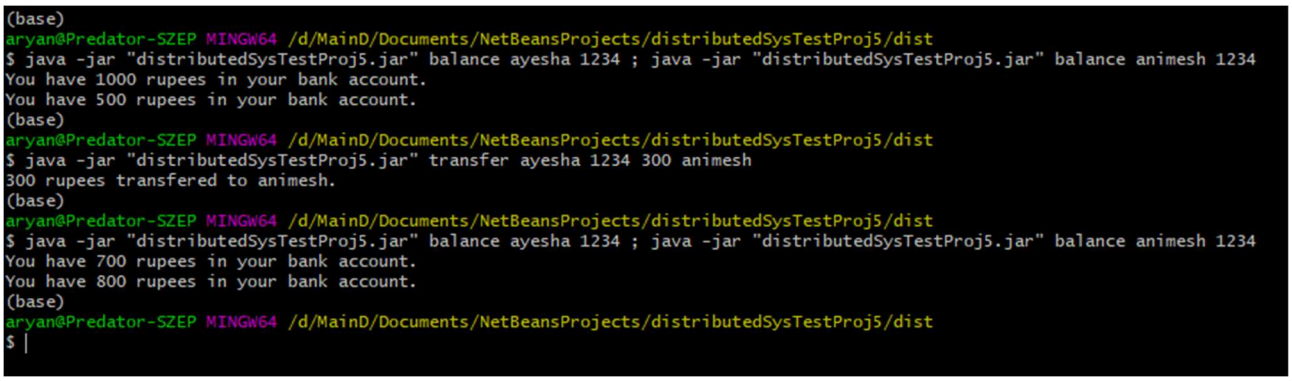


Figure A1.9: Showing change in balance from before after transfer parameter was used to transfer money from account ayesha to account animesh

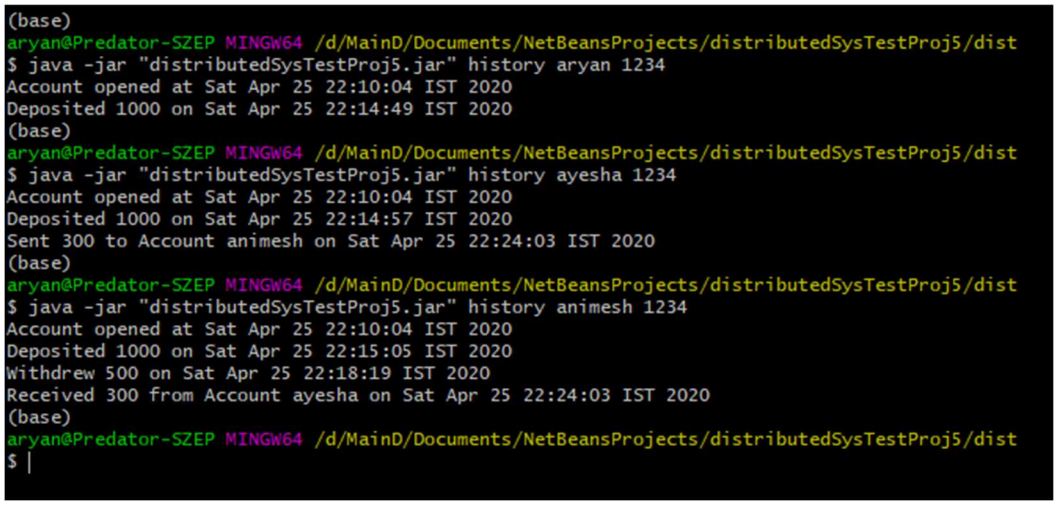


Figure A1.10: Showing transaction history of accounts aryan, ayesha and animesh by using history parameter.

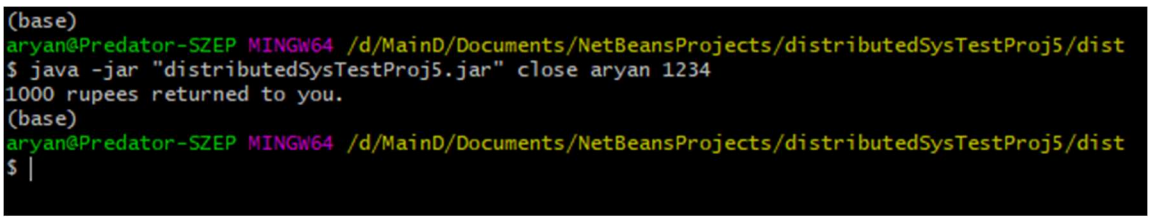


Figure A1.11: Showing how close parameter is used to close a particular account.

Now since all the functionality is shown, we will move to parallel client run test which are done to check for conflict pair situations. These tests are done using JMeter, a testing tool.
Multiple thread groups are created in order to carry out different tests.

Before running these tests on JMeter, all accounts were closed and remade to make sure no transactions have been performed on them. A deposit of Rs. 1000 is kept on balance initially. This is repeated for each test run to understand results better.

Thread Group – History & Balance | ALL will be active during all the threads.

We will be disabling all thread groups other than the one that has to be tested and *Thread Group – History & Balance | ALL*. Thread groups along with bzm - Parallel Controller is used to run the queries together at the same time. View Results gives the result of the query made.

Test case 1:

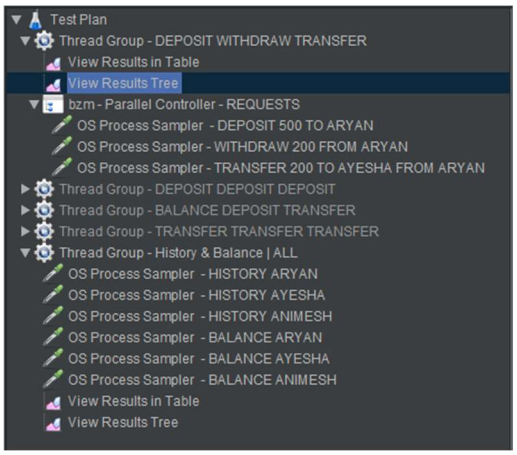


Figure A1.12: Test plan showing deposit withdraw transfer.

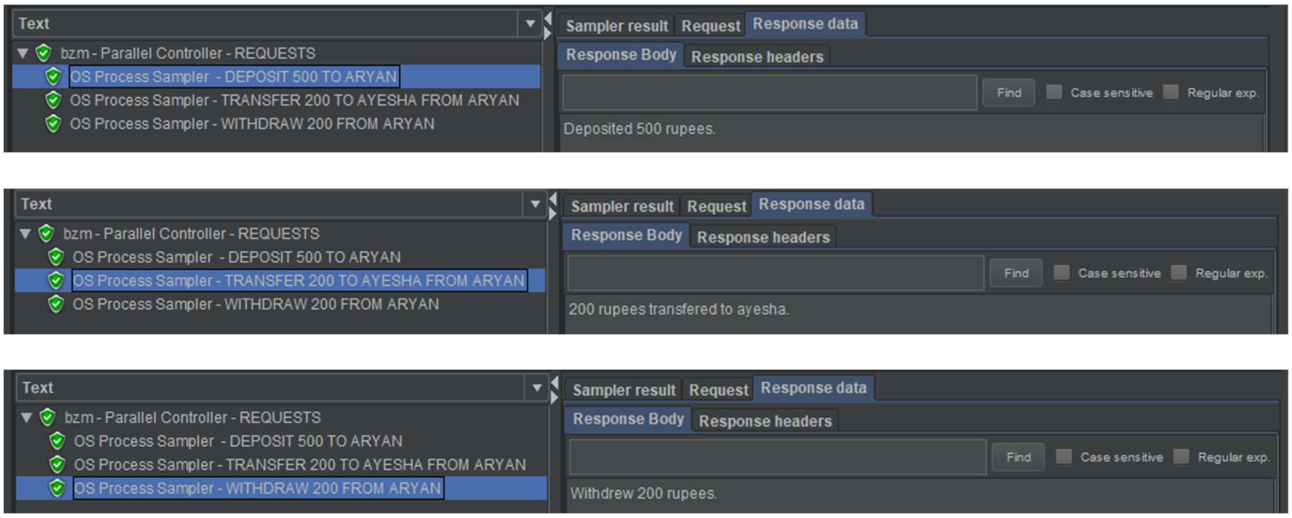
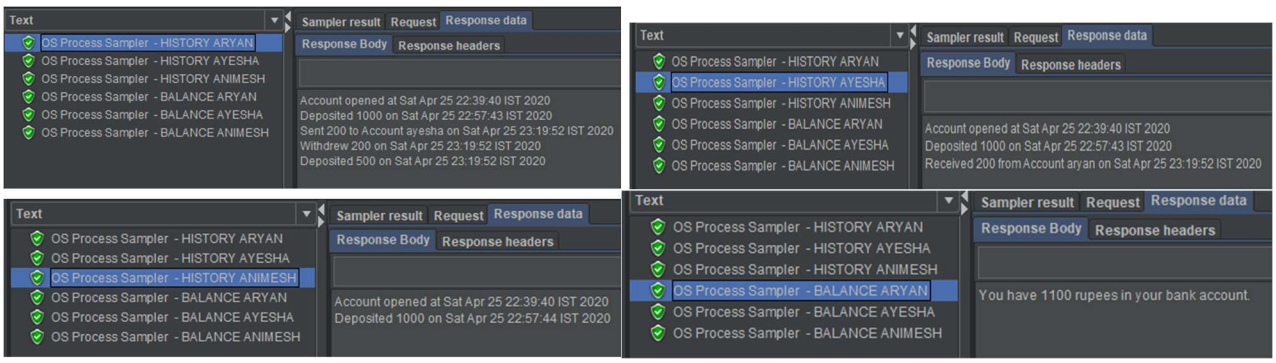


Figure A1.13: Run of Test plan in Figure A1.12 shown



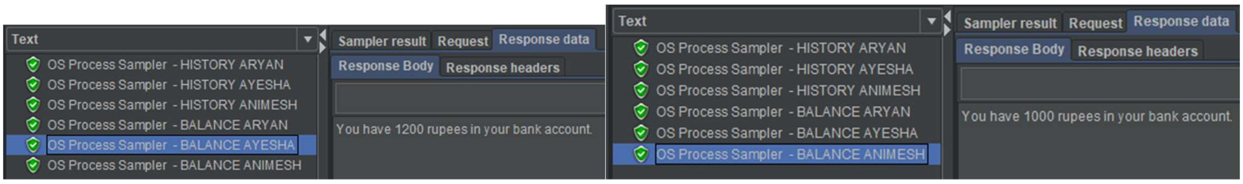


Figure A1.14: History and balance of all accounts after running test plan in Figure A1.12.

Test 1 successful since the calculations were correct and no errors were found.

Test case 2:

Not depositing the accounts with Rs. 1000 in this run.

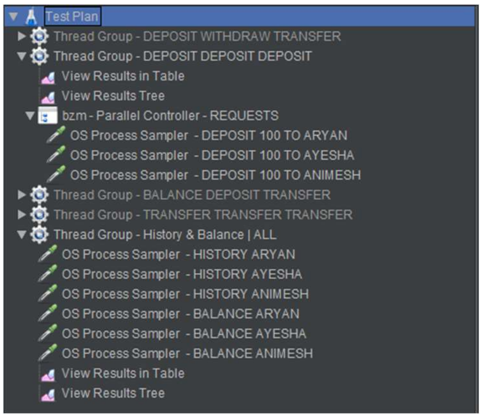


Figure A1.15: Test plan showing deposit deposit deposit.

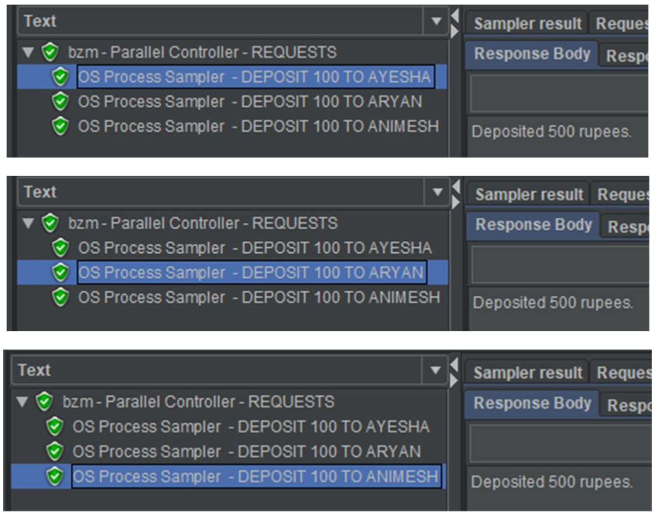


Figure A1.16: Run of Test plan in Figure A1.15 shown

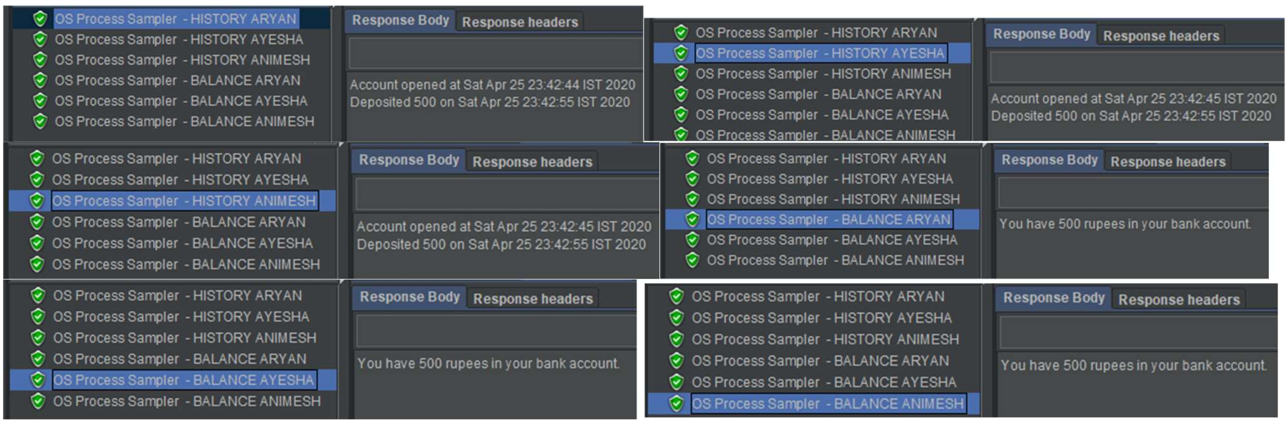


Figure A1.17: History and balance of all accounts after running test plan in Figure A1.15.

Test 2 successful since the calculations were correct and no errors were found.

Test case 3:

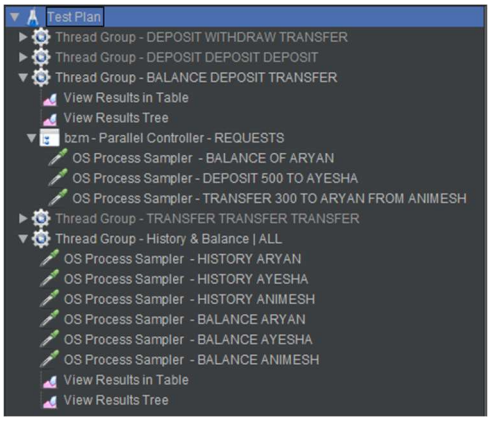


Figure A1.18: Test plan showing balance deposit transfer.

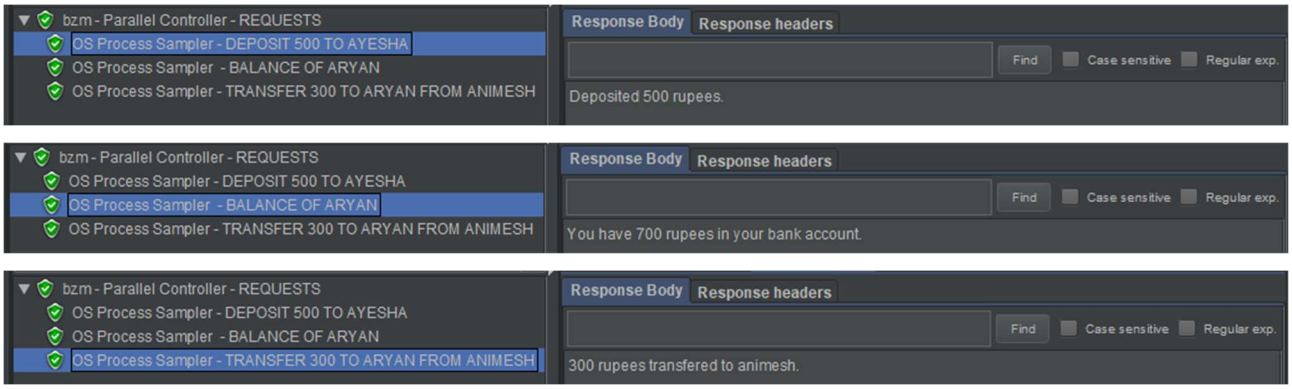


Figure A1.19: Run of Test plan in Figure A1.18 shown

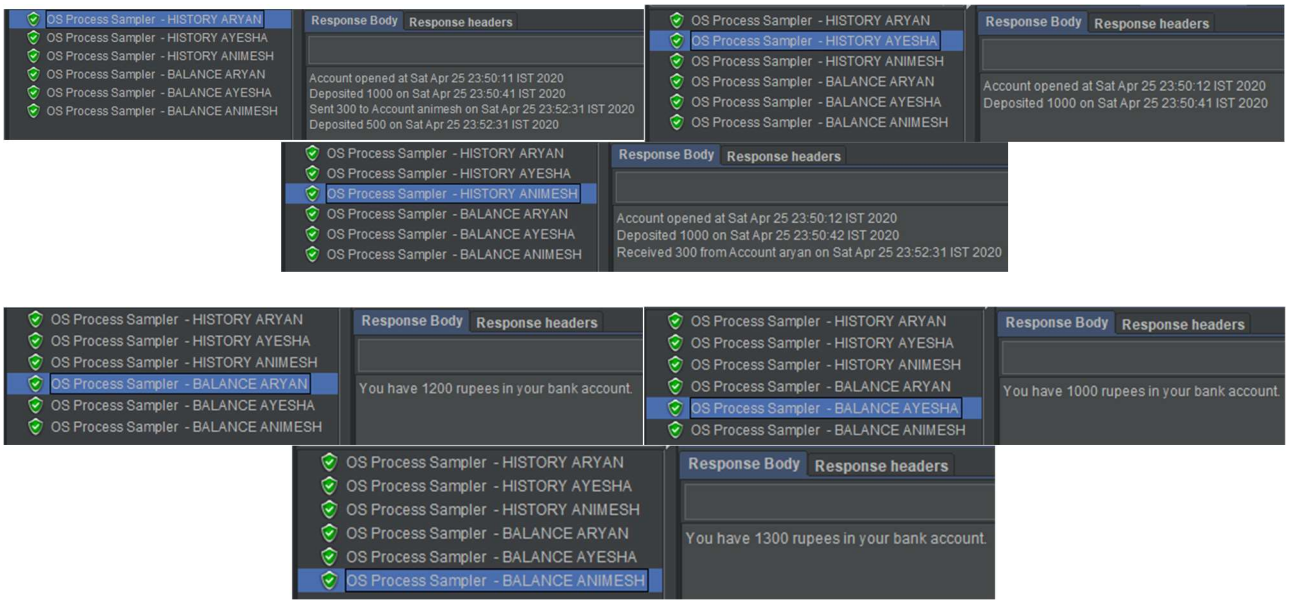


Figure A1.20: History and balance of all accounts after running test plan in Figure A1.18.

Test 3 successful since the calculations were correct and no errors were found.

Test case 4:

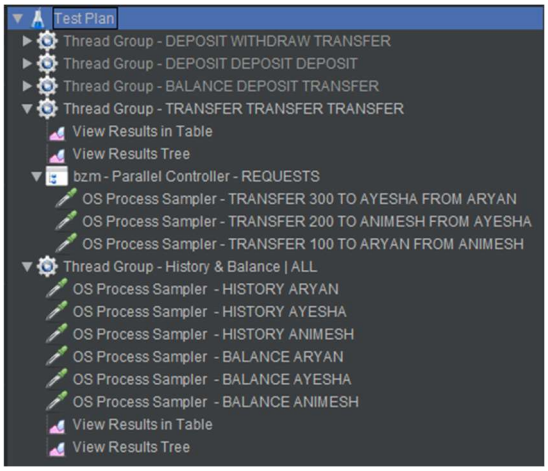


Figure A1.21: Test plan showing transfer transfer transfer.

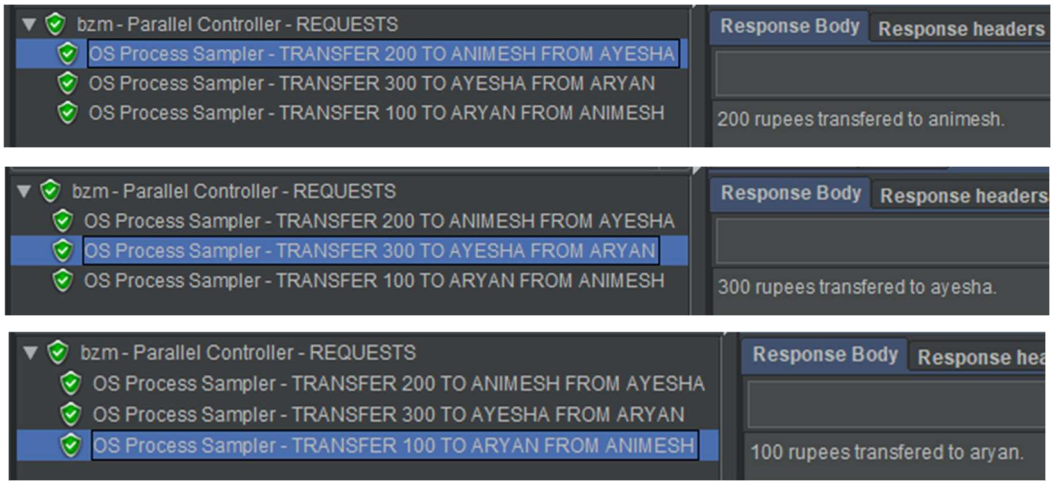


Figure A1.22: Run of Test plan in Figure A1.21 shown

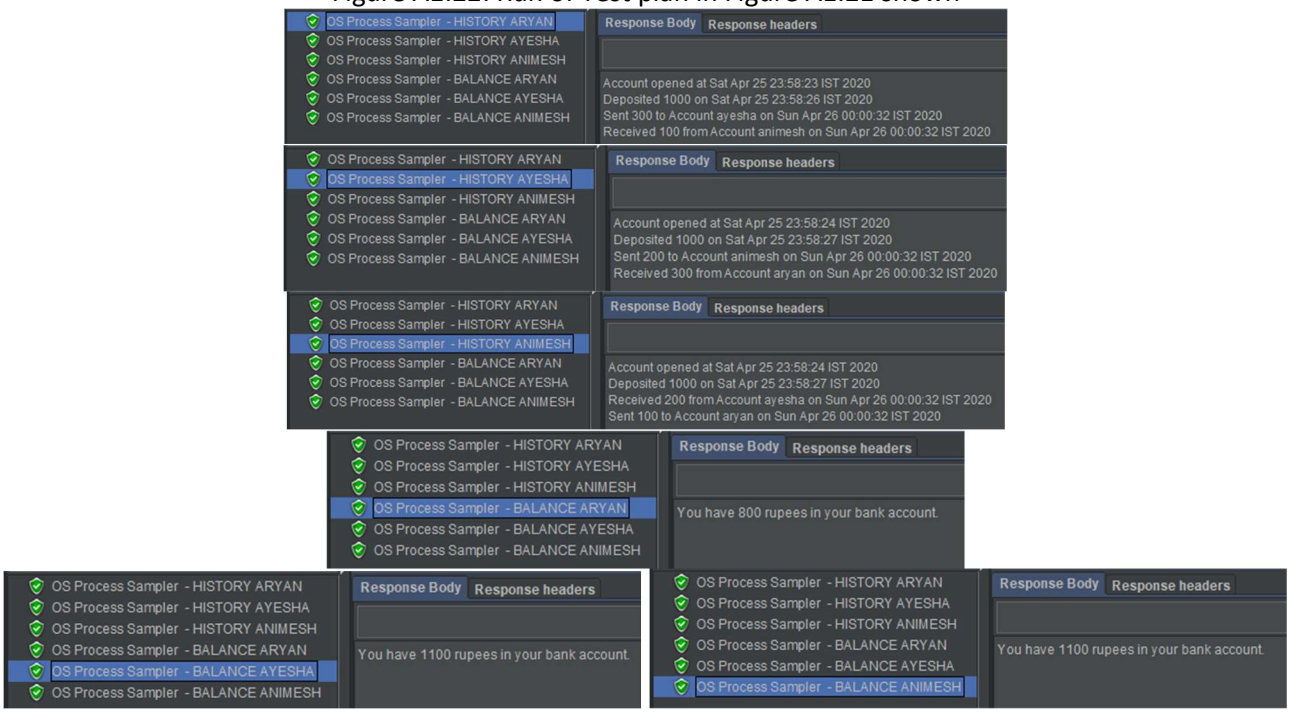


Figure A1.23: History and balance of all accounts after running test plan in Figure A1.21.

Test 4 successful since the calculations were correct and no errors were found.

Question No. B1

Solution to Question No. 2:

B1.1 RPC, the need for it and two sample programs

Remote Procedure Call (RPC) is essentially a protocol that a program can use for requesting a service from another program located in another computer on the network while having all network details abstracted. RPC uses client server model. When RPC is made, the calling program is suspended, procedure parameters are transferred across the network to the program where the procedure is to execute, and the procedure is executed there. When the procedure finishes and produces its results, its results are transferred back to the calling program, where execution resumes as if returning from a regular procedure call.

The main need for RPC is to hide the existence of the network from a program. This makes it so that the message passing system of network is hidden from user. It also improves performance as RPC omits many protocol layers. This improvement is important since program could invoke RPCs often.

Sample programs 1 and 2:

```
1  // server.c file
2
3  #include"rpc/rpc.h"
4  #include"square.h"
5  #include"stdio.h"
6  #include"stdlib.h"
7  #include"math.h"
8
9  square_out *squareproc_1_svc(square_in *inp,struct svc_req *rqstp)
10 {
11     static square_out out;
12     out.res1= inp->arg1 * inp->arg1;
13     return(&out);
14 }
```

Figure B1.1: RPC: Server program 1

```c
//client.c file

#include"errno.h"
#include"rpc/rpc.h"
#include"square.h"
#include"stdio.h"
#include"stdlib.h"
#include"math.h"

int main(int argc,char **argv)
{
    CLIENT *cl;
 square_in in;
 square_out *outp;
 f(argc!=3)
    {
        printf("\n\n error:insufficient arguments!!!");
        exit(-1);
    }

 cl=clnt_create(argv[1],SQUARE_PROG,SQUARE_VERS,"tcp");
 in.arg1=atol(argv[2]);

    if(cl==NULL)
    {
        printf("\nerror:%s",strerror(errno));
        exit(-1);
    }

    if((outp=squareproc_1(&in,cl))==NULL)
    {
        printf("\nerror :%s",clnt_sperror(cl,argv[1]));
        exit(-1);
    }

    printf("\n\n result is : %ld",outp->res1);
    exit(0);
}
```

Figure B1.2: RPC: Client program 1

```c
// square.h file

struct square_in
{
    long arg1; /*input arg*/
};

struct square_out
{
    long res1; /*op result*/
};

program SQUARE_PROG
{
    version SQUARE_VERS
    {
        square_out SQUAREPROC(square_in)=1;     /*proc no=1*/
    }=1;    /*version no*/
}=0x31230000;/  *prog no*/
```

Figure B1.3: Header file for the Client and server programs for RPC 1

```
program DATEPROG {
version DATEVERS {
long BINDATE(void) = 1;} = 1;
} = 0x3012225;
```

Figure B1.4: Header file for the Client and server programs for RPC 2

```
#include stdio.h
#include rpc/rpc.h
#include stdlib.h
#include "date.h"

int main(int argc, char *argv[]) {
    CLIENT *cl;
    char *server;
    long *lres;
    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }
    server = argv[1];
    if ((cl = clnt_create(server, DATEPROG, DATEVERS, "udp")) == NULL) {
        printf("can't establish connection with host %s\n", server);
        exit(2);
    }
    if (( lres = bindate_1(NULL, cl)) == NULL){
        printf(" remote procedure bindate() failure\n");
        exit(3);
    }
    printf("time on host %s = %ld\n", server, *lres);
    clnt_destroy(cl);
    return 0;
    }
```

Figure B1.5: RPC: Client program 2

```
#include stdio.h
#include stdlib.h
#include rpc/rpc.h
#include "date.h"

long * bindate_1() {
    static long timeval; /* must be static */
    timeval = time((long *) 0);
    return (&timeval);
}
```

Figure B1.6: RPC: Server program 2

B1.2 RMI, the need for it and two sample programs

Remote Method Invocation (RMI) is essentially a technique that allows an object residing in a system (JVM) to invoke an object running on another JVM. (Java Virtual Machine) An interface definition language is used in RMI to specify the object's interface. RMI is language specific.

RMI creates a public remote server object that enables client and server-side communications through simple method calls on the server object. This uses client server model as well. The main need of RMI is to

minimize complexity of the whole application, to preserve type safety, to have a distributed garbage collection system and minimize the difference between working with local and remote objects.

Sample programs:

```
1  import java.rmi.*;
2  public interface SubServerInterface extends Remote
3  {
4      public int subtract(int a,int b);
5  }
```

Figure B1.7: Remote Interface 1

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  public class Subtracter extends UnicastRemoteObject implements SubServerInterface{
4      Subtracter() throws RemoteException{
5      super();
6  }
7  public int subtract(int a,int b){
8      return a-b;
9      }
10 }
```

Figure B1.8: Implementation of remote interface 1

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3  public class SubServer {
4      public static void main(String args[]) {
5          try {
6              SubServerInterface subService=new Subtracter();
7              Naming.rebind("SubService",subService);
8          }
9          catch(Exception e) {
10             System.out.println(e);
11         }
12     }
13 }
```

Figure B1.9: RMI: Server program 1

```
1  import java.rmi.*;
2  public class Client {
3      public static void main(String args[]) {
4          try{
5              SubServerInterface st = (SubServerInterface)Naming.lookup("rmi://"+args[0]+"/SubService");
6              System.out.println(st.subtract(30,12));
7          }
8          catch(Exception e) {
9              System.out.println(e);
10         }
11     }
12 }
```

Figure B1.10: RMI: Client program 1

```
1  import java.rmi.*;
2  public interface Power extends Remote
3  {
4      public int power1() throws RemoteException;
5  }
```

Figure B1.11: Remote Interface 2

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.util.Scanner;
4  public class PowerRemote extends UnicastRemoteObject implements Power
5  {
6      PowerRemote() throws RemoteException{
7          super();
8      }
9      public int power1(int z){
10     int z;
11     Scanner sc = new Scanner(System.in);
12     System.out.println("Enter the base number ::");
13     int x = sc.nextInt();
14     System.out.println("Enter the exponent number ::");
15     int y = sc.nextInt();
16     z=y^x;
17     System.out.println(z);
18     }
19 }
```

Figure B1.12: Implementation of remote interface 2

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3  public class MyServer
4  {
5      public static void main(String args[]){
6          try {
7              Power stub=new PowerRemote();
8              Naming.rebind("rmi://localhost:1995/shristee",stub);
9              }
10         catch(Exception e){
11             System.out.println(e);
12             }
13     }
14 }
```

Figure B1.13: RMI: Server program 2

```
1  import java.rmi.*;
2  public class MyClient
3  {
4      public static void main(String args[]){
5          try
6          {
7              Power stub=(Power)Naming.lookup("rmi://localhost:1995/shristee");
8              System.out.println(stub.power1());
9          }
10         catch(Exception e){}
11     }
12 }
```

Figure B1.14: RMI: Client program 2

B1.3 Similarities and differences between RPC and RMI.

Similarities between RPC and RMI:

- RMI and RPC support programming with interfaces
- RMI and RPC offer similar levels of transparency.
- RMI and RPC work on client server model.
- RMI and RPC are made on top of request reply protocols and offer several call semantics.

Differences between RPC and RMI:

- RPC supports procedural programming paradigms whereas RMI supports object-oriented programming paradigms.
- RMI supports systemwide object references whereas RPC doesn't.
- RPC is language neutral whereas RMI is limited to Java.
- RPC supports only primitive data types whereas RMI allows objects to be passed as arguments and return values. When using RPC, programmer must split any compound objects to primitive data types.
- RMI is slower comparatively to RPC since it involves execution of Java Bytecode.
- RMI is easier to program with than RPC.
- RMI allows the use of design patterns since it is object-oriented while RPC does not have this capability.