

ASSIGNMENT

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

Name of the Student	Shikhar singh
Reg. No	17ETCS002168
Semester/Year	06/2017
Course Leader/s	Chaitra S.

Declaration Sheet			
Student Name	Shikhar singh		
Reg. No	17ETCS002168		
Programme	B. Tech	Semester/Year	06/2017
Course Code	CSE301A		
Course Title	Distributed Systems		
Course Date		to	
Course Leader	Chaitra S.		
<p>Declaration</p> <p>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.</p>			
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	

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)				
<div>Signature of First Examiner</div> <div>Signature of Second Examiner</div>				

Declaration Sheet	ii
Contents	iv
List of Figures	5
Question No. 1	6
1.1 Overview:.....	6
1.2 Implementation of banking application to reflect correct and consistent states of account balance when multiple clients initiate transactions:	9
1.3 Testing of the developed banking application:	12
Question No. 2	15
2.1 RPC, the need for it and two sample programs:	15
2.2 RMI, the need for it and two sample programs:	18
2.3 Similarities and Differences between RPC and RMI:.....	22
Bibliography	24
APPENDIX	25

List of Figures

Figure 1: records.txt	9
Figure 2: transaction.txt	9
Figure 3: mutex lock and withdrawal functionality	10
Figure 4: deposit amount and view amount functionality.	11
Figure 5: makefile	12
Figure 6:record file before transaction.	13
Figure 7: record file after transaction	13
Figure 8: server logs for testcase 1.	13
Figure 9: record file before transaction	14
Figure 10: record file after transaction	14
Figure 11: server logs testcase-2	15
Figure 12: RPC "hello world" program	17
Figure 13: RPC "hello world" program	17
Figure 14:RPC "addition of 2 numbers" program	18
Figure 15: RPC "addition of 2 numbers" program	18
Figure 16: RMI "Hello world" program	20
Figure 17: RMI "Hello world" program	20
Figure 18: RMI "Hello world" program	21
Figure 19: RMI "addition of 2 numbers" program	21
Figure 20: RMI "addition of 2 numbers" program	21
Figure 21: RMI "addition of 2 numbers" program	22
Figure 22: RMI "addition of 2 numbers" program	22

Solution to Question No. 1:

1.1 Overview:

We see distributed system in our everyday life, whether it is world wide web or online banking system. Usage of distributed system has grown significantly. Here, we have implemented distributed bank system. We have specifically used distributed system only because it makes the process faster as the work is divided among different system. Every atm is a client and the bank is the server. There is only one server and many clients because in real life also, there is only a single bank and many ATMs. We have provided various remote function which the ATMs can use such as withdrawal of money, transfer of money, view bank statement. But at the same time, when multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. Such problems are called as **concurrency problems**.

These are:

1. Dirty read problems
2. Unrepeatable Read Problems
3. Lost update problems
4. Phantom read problem

To ensure consistency of the system, it is very important to prevent the occurrence of above problems. This can be done using **concurrency control techniques**.

Concurrency control is provided in a database to:

1. enforce isolation among transactions.
2. preserve database consistency through consistency preserving execution of transactions.
3. resolve read-write and write-read conflicts.

Various concurrency control techniques are:

1. Two-phase locking Protocol
2. Time stamp ordering Protocol
3. Multi version concurrency control
4. Validation concurrency control

Here we are using a two-phase locking protocol: -

Locking is an operation which secures: permission to read, OR permission to write a data item. Two phase locking is a process used to gain ownership of shared resources without creating the possibility of deadlock.

The 3 activities taking place in the two phase update algorithm are:

- (i). Lock Acquisition
- (ii). Modification of Data
- (iii). Release Lock

Two phase locking prevents deadlock from occurring in distributed systems by releasing all the resources it has acquired, if it is not possible to acquire all the resources required without waiting for another process to finish using a lock. This means that no process is ever in a state where it is holding some shared resources, and waiting for another process to release a shared resource which it requires. This means that deadlock cannot occur due to resource contention.

Shared lock (S)- this lock type, when imposed, will reserve a page or row to be available only for reading, which means that any other transaction will be prevented to modify the locked record as long as the lock is active. However, a shared lock can be imposed by several transactions at the same time over the same page or row and in that way several transactions can *share* the ability for data reading since the reading process itself will not affect anyhow the actual data.

A transaction in the Two-Phase Locking Protocol can assume one of the 2 phases:

- (i) Growing Phase: In this phase a transaction can only acquire locks but cannot release any lock. The point when a transaction acquires all the locks it needs is called the Lock Point.
- (ii) Shrinking Phase: In this phase a transaction can only release locks but cannot acquire any.

The design techniques used here are as follows:

1. **Socket programming:** Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.
2. **Multi-threading:** Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

- 1) Thread creation is much faster.
 - 2) Context switching between threads is much faster.
 - 3) Threads can be terminated easily
 - 4) Communication between threads is faster.
3. **Thread synchronization (Locking mechanism):** Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where

the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

DESIGN CONCEPT:

Socket programming will be used to simulate a client-server model needed for the banking application. Multithreading is used to handle multiple requests concurrently and thread synchronization is done to prevent concurrency problems as discussed above.

ALGORITHMS:

The algorithm for server-side is as follows:

1. Connecting to a specific port (includes all the server side steps like creating a socket connection, connecting to the port, binding to the port, listening at the port.)
2. Applying a shared lock on the resource.
3. Reading the client records from a file.
4. Cleaning the data
5. computation of new balance depending on the type of transaction, i.e. withdrawal or debit.
6. Printing new balance on the server prompt. (for all the 3 functionalities.)
7. Writing the new balance into record file.
8. Sending Acknowledgement on successful completion of transaction and negative acknowledgement for unsuccessful transactions.
9. Releasing the lock.
10. Flushing the buffer
11. Closing the file
12. Ending connection

The algorithm for client-side is as follows:

1. Reading the data to be sent from a file.
2. Connecting to the port
3. Sending connection request to the server.
4. Sending data when connection is established.
5. Receiving acknowledgement.
6. Printing the acknowledgement in the prompt
7. Closing file
8. Terminating connection.

1.2 Implementation of banking application to reflect correct and consistent states of account balance when multiple clients initiate transactions:

In order to simulate the database system, I am using 2 files, one which holds the record of the user like unique identification number, Name of the user and its Balance. (named record.txt) and another file which simulates user actions like withdraw, debit and view balance (transaction.txt). These actions are shown in terms of 4 argument commands. These arguments are: timestamp, unique identification number, flag argument (to decide whether to withdraw “w”, debit “d” or view the balance “v”.) and the amount by which the balance is to be modified.

```
101 Shikhar 62000
102 Vishal 154900
103 Doron 1400
104 Tyler 31820
```

Figure 1: records.txt

Figure 1 shows records.txt which is a database for all the users.

```
1 5 101 v 0
2 10 102 d 1000
3 11 103 w 500
4
```

Figure 2: transaction.txt

Figure 2 shows transaction.txt which simulates user actions. Here all three type of user actions are shown i.e. view balance, debit amount and withdraw amount respectively.

SERVER SIDE:

1. The server receives queries from customers (ATM clients, Clients/Billing agencies performing online transactions) for operations on accounts.
2. The server has the following functionalities:
 - i) Is able to accept multiple concurrent customer requests (i.e. multi-threaded)
 - ii) It provides locking/protection for access to an account records during shared access (i.e., a user might be depositing money into his account and at the same time a online billing agent might be withdrawing money from the same account).
 - iii) Maintains correctness of records at each record, (i.e., allow withdrawal from an account only if it has sufficient funds etc)

The input file read by the server is shown in figure 1. The format for each line being: account number, name, balance amount.

NOTE: - entire server-side code can be found in the appendix section.

Figure 3 shows the implementation of shared lock mechanism being used to prevent concurrency problems during transactions. When the server successfully finds the user id which is passed by the client, it enables a shared lock (using mutex) which means that any other transaction will be prevented to modify the locked record as long as the lock is active. However, a shared lock can be imposed by several transactions at the same time over the same page or row and in that way several transactions can *share* the ability for data reading since the reading process itself will not affect anyhow the actual data.

```

if (accid == id[i])
{
    pthread_mutex_lock(&mutex2);
    // printf("This account id exists in the database\n");

    //Withdrawal Operation
    if (strcmp(op, "w") == 0)
    {
        printf("%d\t\t\t", balance[i]); //printing previous balance
        printf("Withdrawing amount\t\t");

        if (balance[i] - amount > 0)
        {
            balance[i] = balance[i] - amount;
            printf("%d\t\t\t", balance[i]); //new balance

            //writing the newly calculated balance in the file
            fseek(myfile, size, SEEK_SET); //set the stream pointer i bytes from the start.
            fprintf(myfile, "%d %s %d\n", id[i], name[i], balance[i]);
            printf("Acknowledgement Sent\n\n");

            /* Write a response to the client */
            n = write(newsocket, "ACK: Amount withdrawn\n", 50);
            if (n < 0)
                fprintf(stderr, "Error writing to socket\n");
        }

        else
        {
            //balance insufficient to perform the withdrawal
            printf("Insufficient balance %d\n", balance[i]);
            printf("Acknowledgement Sent\n\n");

            n = write(newsocket, "NACK: Insufficient balance\n", 50);
            if (n < 0)
                fprintf(stderr, "Error writing to socket\n");
        }
    }
}

```

Figure 3: mutex lock and withdrawal functionality

Figure 4 shows the deposit and view amount functionality and the default message i.e. when there is an invalid argument sent by the client, return negative acknowledgement. And at last, releasing the lock to allow other transactions to access the resource. The shared lock applied must allow all the clients to read data from the resource concurrently but perform modifying operations such as withdrawal or debit in a sequential manner as per the 2-phase locking mechanism. This can be verified by looking at the server logs in the testing phase of the problem.

```

//Deposit Operation
else if (strcmp(op, "d") == 0)
{
    printf("%d\t\t\t", balance[i]); //printing previous balance
    printf("Depositing amount\t\t");
    balance[i] = balance[i] + amount;
    printf("%d\t\t\t", balance[i]); //new balance

    //writing the newly calculated balance in the file
    fseek(myfile, size, SEEK_SET); //set the stream pointer i bytes from the start i.e. beginning of line
    fprintf(myfile, "%d %s %d\n", id[i], name[i], balance[i]);
    printf("Acknowledgement Sent\n\n");

    /* Write a response to the client */
    n = write(newsocket, "ACK: Amount deposited\n", 50);
    if (n < 0)
        fprintf(stderr, "Error writing to socket\n");
}

//view transaction
else if (strcmp(op, "v") == 0)
{
    printf("%d\t\t\t", balance[i]); //printing previous balance
    printf("Reviewing amount\t\t");
    printf("%d\t\t\t", balance[i]); //new balance
    printf("Acknowledgement Sent\n\n");

    /* Write a response to the client */
    n = write(newsocket, "ACK: Amount Read\n", 50);
    if (n < 0)
        fprintf(stderr, "Error reading from socket\n");
}

else
{
    printf("Diff value\n");
    /* Write a response to the client */
    n = write(newsocket, "NACK: Invalid transaction type\n", 50);
    if (n < 0)
        fprintf(stderr, "Error writing to socket\n");
}

exist = 1;

//release the lock
pthread_mutex_unlock(&mutex2);
break;

```

Figure 4: deposit amount and view amount functionality.

CLIENT side:

A client issues requests to the server from a transaction based on account numbers.

A Client has the following functionalities:

- i) Issues withdrawal or deposit requests.
- ii) For ease of testing and to make experiments bigger, I have made our clients to issue requests at fixed time intervals. (Hence, the use of timestamps).
- iii) A client can read an input file for transaction information and perform tasks accordingly.

NOTE: the entire client-side code can be viewed in the appendix section.

Furthermore, a Makefile was made to automate the task of writing compilation steps and program execution commands.

The contents of the makefile can be seen in figure below:

```

1 compile:
2   gcc -o server server.c -lm -lpthread
3   gcc -o client client.c
4
5
6 runserver :
7   ./server 8888
8
9
10 runclient:
11   ./client 127.0.0.1 8888 0.2 Transactions.txt
12
13
14 clean :
15   rm server client

```

Figure 5: makefile

The execution steps can be seen in the figure below:

To compile:

```

> make compile
gcc -o server server.c -lm -lpthread
gcc -o client client.c

```

To run server:

```

> make runserver
./server 8888

```

To run client:

```

> make runclient
./client 127.0.0.1 8888 0.2 Transactions.txt

```

1.3 Testing of the developed banking application:

The application was tested as per the parameters set by “ACID” test to check for robustness, consistency, efficiency etc.

In order to simulate multiple client requests, a script was written to run client program for N number of users to perform certain tasks listed in transaction.txt file.

The script file is shown below:

```

for i in {0..10}
do
  ./client 127.0.0.1 8888 0.3 Transactions.txt &
done

```

The testing was done by dividing into different test cases and each test case was conducted for minimum 100 clients on an average and maximum number of clients being 10,000. For the sake of readability, here results are shown for only 11 clients.

Test-case 1: requesting all three transactions, i.e. view balance, withdraw and deposit by the same client using multiple instances. (multiple requests by the same client.)

The arguments in the transaction.txt file were as follows:

5 101 v 0 - view Balance amount from user 101
 10 101 d 1000 - deposit 1000 from user 101
 15 101 w 500 - withdraw 500 from the user 101.

The record file before and after transaction completion (for 11 clients):

```
101 Shikhar 56500
102 Vishal 154900
103 Doron 1400
104 Tyler 31820
```

Figure 6: record file before transaction.

```
101 Shikhar 62000
102 Vishal 154900
103 Doron 1400
104 Tyler 31820
```

Figure 7: record file after transaction

As it can be seen from figure 6 and figure 7, the amount balance next to Shikhar (101) changes after 3 transactions done by 11 client instances parallelly. This was verified with the manual calculations done and the result was found to be correct.

Also , the server log shows the step by step transaction done at each instance of time.

Server log:

```
> make runserver
./server 8888

Connection Established

allocating thread to each client.....
allocation completed.
generating tabulated client request data.....
```

Data	Thread no.	Current Balance	Acc. Status	New Balance	ACK STATUS
5 101 v 0	139738197722880	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738189268736	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738180814592	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738172360448	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738163906304	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738155452160	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738146998016	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738138543872	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738130089728	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738121635584	56500	Reviewing amount	56500	Acknowledgement Sent
5 101 v 0	139738113181440	56500	Reviewing amount	56500	Acknowledgement Sent
10 101 d 1000	139738197722880	56500	Depositing amount	57500	Acknowledgement Sent
12 101 w 500	139738197722880	57500	Withdrawing amount	57000	Acknowledgement Sent
10 101 d 1000	139738189268736	57000	Depositing amount	58000	Acknowledgement Sent
12 101 w 500	139738189268736	58000	Withdrawing amount	57500	Acknowledgement Sent
10 101 d 1000	139738180814592	57500	Depositing amount	58500	Acknowledgement Sent
12 101 w 500	139738180814592	58500	Withdrawing amount	58000	Acknowledgement Sent
10 101 d 1000	139738172360448	58000	Depositing amount	59000	Acknowledgement Sent
12 101 w 500	139738172360448	59000	Withdrawing amount	58500	Acknowledgement Sent
10 101 d 1000	139738163906304	58500	Depositing amount	59500	Acknowledgement Sent
12 101 w 500	139738163906304	59500	Withdrawing amount	59000	Acknowledgement Sent
10 101 d 1000	139738155452160	59000	Depositing amount	60000	Acknowledgement Sent

Figure 8: server logs for testcase 1.

As it can be seen from figure 8, the shared lock is working exactly as needed by letting all the clients access the resource in the reading phase (view balance), then modifying the database based on the 2-phase locking hierarchy. (i.e. modifications done in a serial manner one by one in the order of incoming client requests.)

NOTE: relevant client side logs can be seen in appendix section.

Test-case 2: requesting all three transactions, i.e. view balance, withdraw and deposit by the different clients concurrently. (multiple requests by the different clients.)

The arguments in the transaction.txt file were as follows:

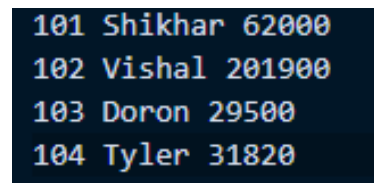
```
5 101 v 0          - view Balance amount from user 101
10 102 d 1000      - deposit 1000 from user 102
15 103 w 500       - withdraw 500 from the user 103.
```

The record file before and after transaction completion (for 11 clients):



```
1 101 Shikhar 62000
2 102 Vishal 190900
3 103 Doron 35000
4 104 Tyler 31820
```

Figure 9: record file before transaction



```
1 101 Shikhar 62000
2 102 Vishal 201900
3 103 Doron 29500
4 104 Tyler 31820
```

Figure 10: record file after transaction

As it can be seen from figure 9 and figure 10, the amount balance is shown changes after 3 transactions done by 11 client instances parallelly. This was verified with the manual calculations done and the result was found to be correct.

Also, the server log shows the step by step transaction done at each instance of time.

SERVER LOGS:

Data	Thread no.	Current Balance	Acc. Status	New Balance	ACK STATUS
5 101 v 0	140231040304896	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140231031850752	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140231023396608	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140231014942464	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140231006488320	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230998034176	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230989580032	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230981125888	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230972671744	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230955763456	62000	Reviewing amount	62000	Acknowledgement Sent
5 101 v 0	140230964217600	62000	Reviewing amount	62000	Acknowledgement Sent
10 102 d 1000	140231040304896	190900	Depositing amount	191900	Acknowledgement Sent
12 103 w 500	140231040304896	35000	Withdrawing amount	34500	Acknowledgement Sent
10 102 d 1000	140231031850752	191900	Depositing amount	192900	Acknowledgement Sent
12 103 w 500	140231031850752	34500	Withdrawing amount	34000	Acknowledgement Sent
10 102 d 1000	140231023396608	192900	Depositing amount	193900	Acknowledgement Sent
12 103 w 500	140231023396608	34000	Withdrawing amount	33500	Acknowledgement Sent
10 102 d 1000	140231014942464	193900	Depositing amount	194900	Acknowledgement Sent
12 103 w 500	140231014942464	33500	Withdrawing amount	33000	Acknowledgement Sent
10 102 d 1000	140231006488320	194900	Depositing amount	195900	Acknowledgement Sent
12 103 w 500	140231006488320	33000	Withdrawing amount	32500	Acknowledgement Sent
10 102 d 1000	140230998034176	195900	Depositing amount	196900	Acknowledgement Sent
12 103 w 500	140230998034176	32500	Withdrawing amount	32000	Acknowledgement Sent
10 102 d 1000	140230989580032	196900	Depositing amount	197900	Acknowledgement Sent
12 103 w 500	140230989580032	32000	Withdrawing amount	31500	Acknowledgement Sent
10 102 d 1000	140230981125888	197900	Depositing amount	198900	Acknowledgement Sent

Figure 11: server logs testcase-2

As it can be seen from figure 11, the shared lock is working exactly as needed by letting all the clients access the resource in the reading phase (view balance), then modifying the database based on the 2-phase locking hierarchy. (i.e. modifications done in a serial manner one by one in the order of incoming client requests.)

NOTE: relevant client-side logs can be seen in appendix section.

Hence, both the testcases showed the expected outcome, and manual calculations showed that the resulting Balance was correct. This was proved for a maximum of 1000 clients requesting concurrently. Thus, the design technique of using 2-phase locking mechanism works effectively in preventing concurrency issues for this application.

Question No. 2

Solution to Question No. 2:

2.1 RPC, the need for it and two sample programs:

Remote Procedure Call is a technique for building distributed systems. Basically, it allows a program on one machine to call a subroutine on another machine without knowing that it is remote. RPC is not a transport protocol: rather, it is a method of using existing communications features in a transparent way. This

transparency is one of the great strengths of RPC as a tool. Because the application software does not contain any communication code, it is independent of

- The particular communications hardware and protocols used
- The operating system used
- The calling sequence needed to use the underlying communications software

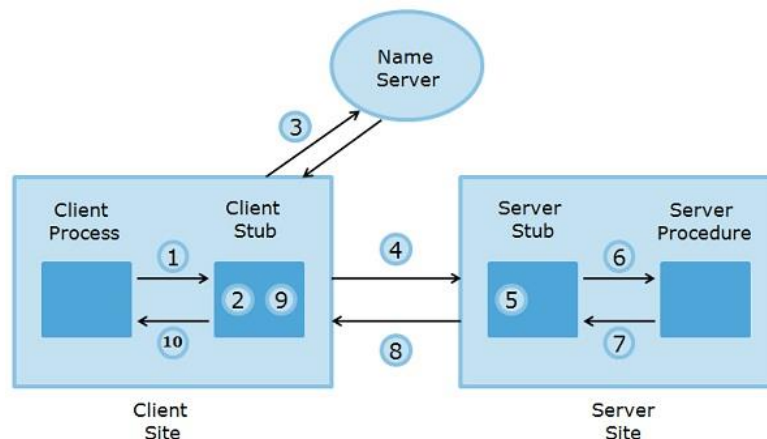
This means that application software can be designed and written before these choices have even been made. Because it takes care of any data reformatting needed, RPC also provides transparency to byte ordering and differences in data representation (real number formats, etc.).

The importance of RPC is as follows:

1. It will decrease the effort to re-write and re-develop the code in remote procedure calls.
2. Message passing mechanism of RPC is hidden from the user.
3. It will support process oriented and thread-oriented models.
4. We can use RPC in distributed environment as well as local environment.

RPC is implemented through the given steps:

- The client process calls the client stub with parameters, and its execution is suspended until the call is completed.
- The parameters are then translated into machine-independent form by marshalling through client stub. Then the message is prepared which contain the representation of the parameters.
- To find the identity of the site the client stub intercommunicate with name server at which remote procedure exists.
- Using blocking protocol the client stub sends the message to the site where remote procedure call exists. This step halt the client stub until it gets a reply.



- The server site receives the message sent from the client side and converts it into machine specific format.
- Now server stub executes a call on the server procedure along with the parameters, and the server stub is discontinued till the procedure gets completed.
- The server procedure returns the generated results to the server stub, and the results get converted into machine-independent format at server stub and create a message containing the results.
- The result message is sent to the client stub which is converted back into machine specific format suitable for the client stub.
- At last client, stub returns the results to the client process.

SAMPLE-CODE 1: hello world.

Server-side:

```
1 #include <stdio.h>
2 #include "hw.h"
3
4 /*
5  * Hello world RPC server -- it just returns the string.
6  */
7
8 char **hw_1_svc(void *a, struct svc_req *req) {
9     static char msg[256];
10    static char *p;
11
12    printf("getting ready to return value\n");
13    strcpy(msg, "Hello world");
14    p = msg;
15    printf("Returning...\n");
16
17    return(&p);
18 }
```

Figure 12: RPC "hello world" program

Client-side:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "hw.h"

/*
 * Simple "hello world" program that demonstrates an rpc call.
 */

main (int argc, char *argv[]) {
    CLIENT *cl;
    char **p;

    if (argc != 2) {
        printf("Usage: client hostname\n");
        exit(1);
    }

    cl = clnt_create(argv[1], HELLO_WORLD_PROG, HELLO_WORLD_VERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }

    printf("Getting ready to call hello world\n");
    p = hw_1(NULL, cl);

    printf("Back from hello world\n");
    if (p == NULL) {
        clnt_perror(cl, argv[1]);
        exit(1);
    }

    printf("Returned string=%s\n", *p);

    return 0;
}
```

Figure 13: RPC "hello world" program

A VERY simple rpc example to start with.

master branch

The RPC server returns the strings "Hello world" when called.

xdr branch Demonstrate the use of xdr.

The RPC server swap the a,b in struct AB, and return it back.

Usage

CMD

> make

> sudo ./server

> ./client localhost

Output

master:

Returned string=Hello world

XDR branch:

Returned a=20, b=10

SAMPLE CODE 2: adding two integers.

Server-side:

```
1  #include "add.h"
2  int *
3  add_1_svc(numbers *argp, struct svc_req *rqstp)
4  {
5      static int result;
6      printf("add(%d,%d) is called\n",argp->a,argp->b);
7      result = argp->a + argp->b;
8      return &result;
9  }
```

Figure 14:RPC "addition of 2 numbers" program

Client-side:

```
1  #include "add.h"
2  void
3  add_prog_1(char *host,int x,int y)
4  {
5      CLIENT *clnt;
6      int *result_1;
7      numbers add_1_arg;
8      #ifndef DEBUG
9      clnt = clnt_create (host, ADD_PROG, ADD_VERS, "udp");
10     if (clnt == NULL) {
11         clnt_pcreateerror (host);
12         exit (1);
13     }
14     #endif /* DEBUG */
15     add_1_arg.a=x;
16     add_1_arg.b=y;
17     result_1 = add_1(&add_1_arg, clnt);
18     if (result_1 == (int *) NULL) {
19         clnt_perror (clnt, "call failed");
20     }
21     else{
22         printf("Result:%d\n", *result_1 );
23     }
24     #ifndef DEBUG
25     clnt_destroy (clnt);
26     #endif /* DEBUG */
27 }
28 int
29 main (int argc, char *argv[])
30 {
31     char *host;
32
33     if (argc < 4) {
34         printf ("usage: %s server_host\n", argv[0]);
35         exit (1);
36     }
37     host = argv[1];
38     add_prog_1 (host,atoi(argv[2]),atoi(argv[3]));
39     exit (0);
40 }
```

Figure 15: RPC "addition of 2 numbers" program

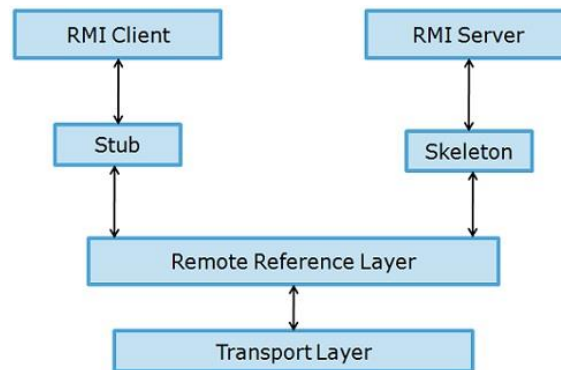
User may enter his password after entering the commands:

./add_client , localhost, 5, 8 are the parameters that are getting passed to the main method in the client program. If a wrong number is passed with parameters it'll show that the typed a syntax is wrong as the definition requires number of parameters to be four. So, giving the required numbers instead of 5 and 8 will yield the desirable sum. It can be seen that add function in the server side is called and result is returned back to the client side.

2.2 RMI, the need for it and two sample programs:

Remote Method Invocation (RMI) is similar to RPC but is language specific and a feature of java. A thread is permitted to call the method on a remote object. To maintain the transparency on the client and server side, it implements remote object using stubs and skeletons. The stub resides with the client and for the

remote object it behaves as a proxy. When a client calls a remote method, the stub for the remote method is called. The client stub is accountable for creating and sending the parcel containing the name of a method and the marshalled parameters, and the skeleton is responsible for receiving the parcel.



The skeleton unmarshals parameters and invokes the desired method on the server. The skeleton marshals the given value (or exceptions) with the parcel and sends it to client stub. The stub reassembles the return parcel and sends it to the client. In Java, the parameters are passed to methods and returned in the form of reference. This could be troublesome for RMI service since not all objects are possibly remote methods. So, it must determine which could be passed as reference and which could not. Java uses process named as **serialization** where the objects are passed as value. The remote object is localised by pass by value. It can also pass an object by reference through passing a remote reference to the object along with the URL of the stub class. Pass by reference restricts a stub for the remote object.

Its Importance are as follows: -

1. **Parallel Computing:** RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.
2. **Easy to Write/Easy to Use:** RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.
3. **Safe and Secure:** RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

Sample-code 1: Hello world

1. **The Remote Interface** The first thing we have to design is the Remote Interface that both Server and Client will implement. The Interface must always be public and extend Remote. All methods described in the Remote interface must list Remote Exception in their throws clause. Our RMI Interface has only one method; it receives a String parameter and returns String.

```

1 package com.mkyong.rmiinterface;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
6 public interface RMIInterface extends Remote {
7
8     public String helloTo(String name) throws RemoteException;
9
10 }

```

Figure 16: RMI "Hello world" program

2. The Server Our Server extends UnicastRemoteObject and implements the Interface we made before. In the main method we bind the server on localhost with the name "MyServer".

```

1 package com.mkyong.rmiserver;
2
3 import java.rmi.Naming;
4 import java.rmi.RemoteException;
5 import java.rmi.server.UnicastRemoteObject;
6
7 import com.mkyong.rmiinterface.RMIInterface;
8
9 public class ServerOperation extends UnicastRemoteObject implements RMIInterface {
10
11     private static final long serialVersionUID = 1L;
12
13     protected ServerOperation() throws RemoteException {
14
15         super();
16     }
17
18
19     @Override
20     public String helloTo(String name) throws RemoteException {
21
22         System.err.println(name + " is trying to contact!");
23         return "Server says hello to " + name;
24     }
25
26
27     Run | Debug
28     public static void main(String[] args) {
29
30         try {
31
32             Naming.rebind("//localhost/MyServer", new ServerOperation());
33             System.err.println("Server ready");
34
35         } catch (Exception e) {
36
37             System.err.println("Server exception: " + e.toString());
38             e.printStackTrace();
39
40         }
41     }
42
43 }

```

Figure 17: RMI "Hello world" program

3. The Client: Finally, we create the Client. To "find" the Server, the Client uses an RMI Interface Object that "looks" for a reference for the remote object associated with the name we pass as parameter. With that RMI Interface object we can now talk to the Server and receive responses.

```

1 package com.mkyong.rmiClient;
2
3 import java.net.MalformedURLException;
4 import java.rmi.Naming;
5 import java.rmi.NotBoundException;
6 import java.rmi.RemoteException;
7
8 import javax.swing.JOptionPane;
9
10 import com.mkyong.rmiinterface.RMIInterface;
11
12 public class ClientOperation {
13     private static RMIInterface look_up;
14
15     Run | Debug
16     public static void main(String[] args)
17         throws MalformedURLException, RemoteException, NotBoundException {
18
19         look_up = (RMIInterface) Naming.lookup("//localhost/MyServer");
20         String txt = JOptionPane.showInputDialog("What is your name?");
21
22         String response = look_up.helloTo(txt);
23         JOptionPane.showMessageDialog(null, response);
24     }
25 }

```

Figure 18: RMI "Hello world" program

SAMPLE-CODE 2: Sum of two numbers:

1. create the Remote interface.

```

1 import java.rmi.*;
2 public interface rmiInterface extends Remote
3 {
4     double add (double d1, double d2) throws Exception;
5 }

```

Figure 19: RMI "addition of 2 numbers" program

2. The rules for the Remote interface are as follows:
 1. The interface must extend java.rmi.Remote
 2. The method signatures for remote methods need to be defined to throw a Remote Exception
- The second step is to implement the interface.

```

import java.rmi.*;
import java.rmi.server.*;
public class rmiImpl extends UnicastRemoteObject
    implements rmiInterface
{
    public rmiImpl () throws RemoteException
    {
    }

    public double add (double d1, double d2) throws Exception
    {
        return d1+d2;
    }
}

```

Figure 20: RMI "addition of 2 numbers" program

3. code a driver program, which will create an instance of the class, and register it with the rmiregistry.

```

1 public static void main (String args [])
2 {
3     try
4     {
5         rmiImpl obj=new rmiImpl ();
6         Naming.rebind ("//rmi", rmiobj);
7
8     }
9     catch (Exception e)
10    {
11    }
12
13    System.out.println ("An error occurred trying to "+"
14                        "bind the object to the registry.");
15 }

```

Figure 21: RMI "addition of 2 numbers" program

4. implement the client.

```

1 import java.rmi.*;
2 import java.io.*;
3 public class ConfusedClient
4 {
5     Run | Debug
6     public static void main (String args [])
7     {
8         BufferedReader reader;
9
10        try
11        {
12            String url="//"+args[0]+"rmi";
13            rmiInterface rmiinter= (rmiInterface) Naming.lookup(url);
14            System.out.println ("\n The first no is"+args[1]);
15            System.out.println ("\n The second no is"+args[2]);
16            double d1, d2;
17            d1=Double.valueOf (args [1]).doubleValue ();
18            d2=Double.valueOf (args [2]).doubleValue ();
19            System.out.println ("\n The sum of two values"+rmiinter.add (d1, d2));
20        }
21        catch (Exception e)
22        {
23            System.out.println ("An error occurred.");
24        }
25    }
26 }

```

Figure 22: RMI "addition of 2 numbers" program

2.3 Similarities and Differences between RPC and RMI:

Similarity between RPC and RMI: -

1. Both support programming with interfaces
2. Both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once
3. Both offer a similar level of transparency- that is, local and remote call employ the same syntax.

Key Differences Between RPC and RMI:

1. RPC supports procedural programming paradigms thus is C based, while RMI supports object-oriented programming paradigms and is java based.
2. The parameters passed to remote procedures in RPC are the ordinary data structures. On the contrary, RMI transmits objects as a parameter to the remote method.
3. RPC can be considered as the older version of RMI, and it is used in the programming languages that support procedural programming, and it can only use pass by value method. As against, RMI facility is devised based on modern programming approach, which could use pass by value or reference. Another advantage of RMI is that the parameters passed by reference can be changed.

4. RPC protocol generates more overheads than RMI.
5. The parameters passed in RPC must be “**in-out**” which means that the value passed to the procedure and the output value must have the same datatypes. In contrast, there is no compulsion of passing “**in-out**” parameters in RMI.
6. In RPC, references could not be probable because the two processes have the distinct address space, but it is possible in case of RMI.

1. https://www.cs.uct.ac.za/mit_notes/database/htmls/chp13.html#shared-and-exclusive-locks
2. <https://www.sqlshack.com/locking-sql-server/>
3. <https://techdifferences.com/difference-between-rpc-and-rmi.html>

SERVER-SIDE SOURCE CODE:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <math.h>
#include <pthread.h>
#include </usr/include/semaphore.h>
#define MAXDATASIZE 256

void *connhandler(void *);

char* buffer[MAXDATASIZE];
int lines = 0;
char* records[MAXDATASIZE];
int id[MAXDATASIZE], balance[MAXDATASIZE];
char* name[MAXDATASIZE];

int sockfd, newsockfd, portno, *new_sock;
struct sockaddr_in serveraddr, clientaddr;

FILE* myfile;
int i = 0;
int accid, amount;
char op[3];
char contents[MAXDATASIZE];
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
int x = 0;

size_t len = 0;

int main(int argc, char** argv)
{
    if (argc < 2)
    {
        fprintf(stderr, "Usage: server <port_number>\n");
        exit(1);
    }

    //reading from records of accounts
    char ch;
    myfile = fopen("Records.txt", "r+");
    if (myfile == NULL)
    {
        printf("Error opening file\n");
    }

    do
    {
        ch = fgetc(myfile);
        if (ch == '\n')
            lines++;
    } while (ch != EOF);

    //printf("%d\n", lines);
    rewind(myfile);

    //fetching individual data from each record
    for (i = 0; i < lines; i++)
    {
        records[i] = NULL;
    }

```

```

len = 0;
getline(&records[i], &len, myfile);
//printf("%s\n", records[i]);
char* p;
p = strtok(records[i], " ");

if (p)
{
    id[i] = atoi(p);
    //printf("%s %d\n", p, id[i]);
}

p = strtok(NULL, " ");

if (p)
{
    name[i] = p;
    //printf("%s\n", p);
}

p = strtok(NULL, " ");

if (p)
{
    balance[i] = atoi(p);
    // printf("%s\n", p);
}
}

/* First call to socket() function */
socketfd = socket(AF_INET, SOCK_STREAM, 0);
if (socketfd < 0)
    fprintf(stderr, "Error creating a socket\n");

/* Initialize socket structure */
bzero((char*)&serveradd, sizeof(serveradd));

portno = atoi(argv[1]);
serveradd.sin_family = AF_INET;
serveradd.sin_addr.s_addr = INADDR_ANY;
serveradd.sin_port = htons(portno);

/* Now bind the host address using bind() call.*/
if (bind(socketfd, (struct sockaddr*)&serveradd, sizeof(serveradd)) < 0)
{
    fprintf(stderr, "Error binding socket\n");
}

/* Now start listening for the clients, here process will
   go in sleep mode and will wait for the incoming connection
*/
listen(socketfd, 5);

socklen_t clientlen = sizeof(clientaddr);

pthread_mutex_init(&mutex2, NULL);

/* Accept actual connection from multiple clients */
while (newsockfd = accept(socketfd, (struct sockaddr*)&clientaddr, &clientlen))
{
    if (newsockfd < 0)
    {
        fprintf(stderr, "Error accepting client request\n");
    }
    else
    {
        printf("\n\nConnection Established\n\n");
        printf("allocating thread to each client.....");
        printf("\nallocation completed.");
        printf("\ngenerating tabulated client request data.....\n\n\n");
    }
}

```

```

pthread_t p_thread;
new_sock = malloc(1);
*new_sock = newsockfd;

//create a new thread for each client
if(pthread_create(&p_thread , NULL , connhandler , (void*) new_sock) < 0)
{
    printf("Could not create thread\n");
    return 1;
}
}

//close the connection socket
close(newsockfd);
pthread_exit(NULL);

//close the file
fclose(myfile);

//close the server socket
close(socketfd);
return 0;
}

//called for each client thread
void *connhandler(void *socket_desc)
{
    //Get the socket descriptor
    x++;
    //printf("Client %d\n", x);

    /* If connection is established then start communicating */
    bzero(buffer, MAXDATASIZE);
    int newsocket=*(int *)socket_desc;
    //receiving from client (transactions)
    // printf("Thread :%ld\n", pthread_self()); // print
    bzero(contents, MAXDATASIZE);
    int n=0;
    printf("Data\t\t\tThread no.\t\t\tCurrent Balance\t\tAcc. Status\t\t\tNew Balance\t\tACK STATUS\n");
    printf("-----\n");
    while(n=read(newsocket, contents, MAXDATASIZE))
    {
        if (n < 0)
        {
            fprintf(stderr, "Error in receiving data\t");
            exit(1);
        }

        if(strlen(contents)==0)
        {
            continue;
        }

        printf("%s\t\t", contents); //Data received from client: print

        printf("%ld\t\t\t", pthread_self()); //Thread number print

        //separating account id, transaction to be performed and the amount from the received data
        char* p;
        p = strtok(contents, " ");
        p = strtok(NULL, " ");

        if (p)
        {
            accid = atoi(p);
            //printf("%d\n", accid);
        }

        p = strtok(NULL, " ");

        if (p)
        {
            strcpy(op, p);

```

```

        //printf("%s\n", op);
    }

    p = strtok(NULL, " ");

    if (p)
    {
        amount = atoi(p);
        //printf("%d\n", amount);
    }

    int size = 0;

    int exist = 0;

    //check whether id in the requested transaction exists in the database
    for (i = 0; i < lines; i++)
    {
        if (i == 0)
        {
            size = 0;
        }
        else
        {
            size += (fFloor(log10(abs(id[i - 1])))) + 1 + strlen(name[i - 1]) + (fFloor(log10(abs(balan
ce[i - 1])))) + 1) + 3;
        }
        if (accid == id[i])
        {
            pthread_mutex_lock(&mutex2);
            // printf("This account id exists in the database\n");

            //Withdrawal Operation
            if (strcmp(op, "w") == 0)
            {
                printf("%d\t\t\t", balance[i]); //printing previous balance
                printf("Withdrawing amount\t\t");

                if (balance[i] - amount > 0)
                {
                    balance[i] = balance[i] - amount;
                    printf("%d\t\t\t", balance[i]); //new balance

                    //writing the newly calculated balance in the file
                    fseek(myfile, size, SEEK_SET); //set the stream pointer i bytes from the start.
                    fprintf(myfile, "%d %s %d\n", id[i], name[i], balance[i]);
                    printf("Acknowledgement Sent\n\n");

                    /* Write a response to the client */
                    n = write(newsocket, "ACK: Amount withdrawn\n", 50);
                    if (n < 0)
                        fprintf(stderr, "Error writing to socket\n");

                }

                else
                {
                    //balance insufficient to perform the withdrawal
                    printf("Insufficient balance %d\n", balance[i]);
                    printf("Acknowledgement Sent\n\n");

                    n = write(newsocket, "NACK: Insufficient balance\n", 50);
                    if (n < 0)
                        fprintf(stderr, "Error writing to socket\n");

                }
            }

            //Deposit Operation
            else if (strcmp(op, "d") == 0)
            {
                printf("%d\t\t\t", balance[i]); //printing previous balance
                printf("Depositing amount\t\t");
            }
        }
    }
}

```

```

        balance[i] = balance[i] + amount;
        printf("%d\t\t\t", balance[i]); //new balance

        //writing the newly calculated balance in the file
        fseek(myfile, size, SEEK_SET); //set the stream pointer i bytes from the start i.e. b
eginning of line

        fprintf(myfile, "%d %s %d\n", id[i], name[i], balance[i]);
        printf("Acknowledgement Sent\n\n");

        /* Write a response to the client */
        n = write(newsocket, "ACK: Amount deposited\n", 50);
        if (n < 0)
            fprintf(stderr, "Error writing to socket\n");
    }

    //view transaction
    else if (strcmp(op,"v") == 0)
    {
        printf("%d\t\t\t",balance[i]); //printing previous balance
        printf("Reviewing amount\t\t");
        printf("%d\t\t\t", balance[i]); //new balance
        printf("Acknowledgement Sent\n\n");

        /* Write a response to the client */
        n = write(newsocket, "ACK: Amount Read\n", 50);
        if (n < 0)
            fprintf(stderr, "Error reading from socket\n");
    }

    else
    {
        printf("Diff value\n");
        /* Write a response to the client */
        n = write(newsocket, "NACK: Invalid transaction type\n", 50);
        if (n < 0)
            fprintf(stderr, "Error writing to socket\n");
    }

    exist = 1;

    //release the lock
    pthread_mutex_unlock(&mutex2);
    break;
}

}

if (exist == 0)
{
    //the requested id is invalid
    printf("This account id does not exist in the rocrds\n\n");
    n = write(newsocket, "NACK\n", 4);
    if (n < 0)
        fprintf(stderr, "Error writing to socket\n");
}

}

//flush the contents of the buffer into the file
fflush(myfile);

return 0;
}

```

CLIENT-SIDE SOURCE CODE:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

```

```

#include <time.h>
#include <math.h>

#define MAXFILE 500
int main(int argc, char **argv)
{
    if(argc < 4)
    {
        fprintf(stderr, "Usage client <hostname> <port number> <timestamp> <filename>\n");
        exit(1);
    }

    char buffer[MAXFILE];
    int x=0;
    char ch;
    char line[200];
    char *filename;
    filename=argv[4];

    //open the filename received from the command line
    FILE *myfile = fopen(filename, "r");
    if(myfile==NULL)
    {
        printf("Error opening file\n");
    }

    struct hostent *server;
    int portno;
    double sec;
    //extract values from command line arguments
    portno = atoi(argv[2]);
    sec = atof(argv[3]);
    server = gethostbyname(argv[1]);
    if(server == NULL)
    {
        fprintf(stderr, "No such host exists\n");
        exit(1);
    }

    struct sockaddr_in server_addr;

    server_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&server_addr.sin_addr.s_addr,server->h_length);
    server_addr.sin_port = htons(portno);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if(sockfd < 0)
    {
        fprintf(stderr, "Socket not formed \n");
        exit(0);
    }

    double time;
    double initial;
    //connect to server socket
    if(connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
    {
        fprintf(stderr, "Error connecting \n");
    }

    x++;
    memset(buffer, 0, MAXFILE);
    int size=0;
    char string[MAXFILE];
    int transactions=0;

    //get each transaction and send it one by one
    while (fgets(buffer, MAXFILE, myfile) != NULL)
    {
        transactions++;
        initial=initial+1;
    }
}

```

```

char *p;
p=strtok(buffer,"\n");
if(p)
{
    strcpy(buffer,p);
}

size+=strlen(buffer);
//printf("%s %lu %d buffer \n", buffer,strlen(buffer),size);

char record[256];
strcpy(record,buffer);

//checking timestamp before sending
p=strtok(record," ");
if(p)
{
    time=atoi(p);
}
//delay based on the timestamp provided in the file and the time already spent

while(initial!=time)
{
    if(initial>time)
    {
        sleep((initial-time)*sec);
        initial=initial+1;
        //printf("wait%f\n", (initial-time)*sec);
    }

    else
    {
        sleep((time-initial)*sec);
        initial=initial+1;
    }
}

int n;
if(initial==time)
{
    //initial=time;
    printf("Sending----->> %s\n\n",buffer);

    n = write(socketfd, buffer, sizeof(buffer));
    if(n < 0)
    {
        fprintf(stderr, "Error with writing to socket\n");
        exit(1);
    }
    bzero(buffer, MAXFILE);
    //read the acknowledgement received from server
    n = read(socketfd, buffer, MAXFILE);
    printf("Message Received : %s\n",buffer);
}

}
close(socketfd);
exit(0);

return 0;
}
//IP addr - 172.17.7.74

```

CLIENT LOGS FOR TESTCASE-1 AND TESTCASE-2 IN 1.3:

TEST-CASE1:

```
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Message Received : ACK: Amount Read  
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Sending----->> 5 101 v 0  
  
Message Received : ACK: Amount Read  
Sending----->> 10 101 d 1000  
  
Message Received : ACK: Amount deposited  
Sending----->> 12 101 w 500  
  
Message Received : ACK: Amount withdrawn  
Sending----->> 10 101 d 1000  
  
Message Received : ACK: Amount deposited  
Sending----->> 12 101 w 500  
  
Message Received : ACK: Amount withdrawn
```

TEST-CASE2:

```
> bash script.sh
[~] > ~/documents/Multi-User-Bank-Ac
Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 5 101 v 0

Message Received : ACK: Amount Read

Sending-----> 10 102 d 1000
```