
# UNIVERSITY OF MARYLAND BALTIMORE COUNTY

## DEPARTMENT OF COMPUTER SCIENCE

**ADVANCED OPERATING SYSTEMS**

**CMSC 621**

**PROJECT 3**

**SUBMITTED BY:**

**SUSHMITHA MANJUNATHA**

# ABSTRACT

In this project, I have tried to implement a distributed banking service that permits multiple concurrent operations and exhibits simple fault tolerance. You should run 3 servers, each as a separate process on your machine. Each server should be capable of handling multiple simultaneous requests, and any account should be accessible from any server whether or not it was created on that server. Additionally, the group of servers should tolerate the loss of any one server in the group without impacting the overall service. You can kill a process to emulate a crash. What this means is that you must replicate the data between servers. Servers should maintain just enough state as to complete the transactions. For locking, you can use any locking mechanism. For atomicity, you should use the two-phase commit protocol. Write your server and client in C++.

# SYSTEM DESIGN

## Description of the functions used:

### In-built functions used:

**argc -** It is the number of arguments passed into the program from the command line, including the name of the program.

**argv[]** - The array of character pointers is the listing of all the arguments provided in the command line.

**stderr** -Standard error is an output stream typically used by programs to output error messages or diagnostics. It is a stream independent of standard output and can be redirected separately.

**gethostbyname()** – This function returns a structure of type *hostent* for the given host *name*. Here *name* is either a hostname or an IPv4 address in standard dot notation.

**atoi** - Convert string to integer.

**atof** - Convert string to double.

**sockaddr_in** - The basic structure for all syscalls and functions that deal with internet addresses.

**socket(domain,type,protocol)** - The function socket() creates an endpoint for communication and returns a file descriptor for the socket. socket() takes three arguments: domain, type and protocol.

**AF_INET**- Indicates network protocol IPv4 (IPv4-only).

**SOCK_STREAM** – It is reliable stream-oriented service .

**bind()** - It assigns a socket to an address. When a socket is created using socket(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts. bind() takes three arguments:

sockfd, a descriptor representing the socket to perform the bind on.

my_addr, a pointer to a sockaddr structure representing the address to bind to.

addrlen, a socklen_t field specifying the size of the sockaddr structure.

Bind() returns 0 on success and -1 if an error occurs.

**listen()** – After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (SOCK_STREAM, SOCK_SEQPACKET). listen() requires two arguments:

sockfd, a valid socket descriptor.

backlog, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

**ifstream -** Stream class to read from files

**fstream -** Stream class to both read and write from/to files.

**htons**() - This function converts the unsigned short integer *hostshort* from host byte order to network byte order.

**filename.c_str()**- If *filename* is a variable of type *string*, we can obtain the corresponding C-style string by calling the function *filename.c_str().*

**bzero(buff,n)** - The bzero() function erases the data in the *n* bytes of the memory starting at the location pointed to by buff, by writing zeroes (bytes containing '\0') to that area.

**sprint(char *str ,const char *format)** - Composes a string with the same text that would be printed if *format* was used on printf, but instead of being printed, the content is stored as a *C string* in the buffer pointed by *str*.

**write(int *fd*, const void **buf*, size_t *count*) -** It writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

**read(int *fd*, void **buf*, size_t *count*) -** attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

**sockfd -** It is the socket descriptor returned by socket(). serv_addr is pointer to struct sockaddr that contains information on destination IP address and port. addrlen is set to sizeof(struct sockaddr)

**sleep() –** It makes the calling thread sleep until specified seconds have elapsed or a signal arrives which is not ignored.

**accept()** - When an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using the accept() function. The accept() function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

sockfd, the descriptor of the listening socket that has the connection queued.

cliaddr, a pointer to a sockaddr structure to receive the client's address information.

addrlen, a pointer to a socklen_t location that specifies the size of the client address structure passed to accept(). When accept() returns, this location indicates how many bytes of the structure were actually used.

The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs.

**pthread_attr_init(&tattr)-** This function initializes the thread attributes object pointed to by *attr* with default attribute values.

**pthread_attr_setdetachstate((&tattr, int detachstate)** – This function sets the detach state attribute of the thread attributes object referred to by t*attr* to the value specified in *detachstate*. The detach state attribute determines whether a thread created using the thread attributes object t*attr* will be created in a joinable or a detached state.

**pthread_create()** - This function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

**pthread_mutex_init(&mutex,NULL)** - This function shall initialize the mutex referenced by *mutex* with attributes specified by*attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

**pthread_mutex_lock(&mutex)** - The mutex object referenced by *mutex* is locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

**pthread_mutex_unlock(&mutex)** - This function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute.

## Structures Used:

**records -** It stores the records of the accounts in the bank-server. It has attributes like account-number and balance.

**How to run the code:**

1.g++ backendS.cpp -o backendS -lpthread

2.g++ frontendS.cpp -o frontendS -lpthread

3.g++ clientS.cpp -o clientS

Backend server:

4. ./backendS 9090 1

5. ./backendS 9091 2

6. ./backendS 9092 3

FrontendServer side

7. ./frontend localhost 8888 9090 9091 9092

Client:

8. ./clientS <localhost> <clientportnumber>

## Frontend server

1.Three seperate buffers are being used for each of the 3 servers.

2.Accepts 6 command line aruments i.e client_port_number,and port_numbers of all 3 servers along with the file name.

3.flag  "NW" is being used to keep track of number of not working servers initially it will be 0 assuming alls server works.

4.(setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse, sizeof(reuse)) - this function is being used SO_REUSEADDR Specifies that the rules used in validating addresses supplied to bind() should allow reuse of local addresses, if this is supported by the protocol. This option takes an int value. This is a Boolean option.

5.Now socket connection is made with 3 servers.

6.Connection is made with the client process

7.pthread_create( &t , NULL ,  handler_function , (void*) new_socketfd) < 0) - is used MULTITHREADING is done for handling each client request.

8.handler_function() - handles 2phase commit and locking scheme

9.reads multiple requests from clients

10**.pthread_mutex_lock(&mutex)- mutexs** are used for locking

11.Timeout values are set using "struct timeval"

11.Based on value of NW flag, COMMAND will be sent to the server

12**.(setsockopt (socketfd1, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout_values,sizeof(struct timeval))--** is to use setsockopt() to set the socket's SO_RCVTIMEO option.

13.OK message is being read from all 3 servers.

14.when OK has been received from all the servers check if response has been received from all of them if not, then think of maintaining data consistency

15.If all servers are ready to commit,send global commit to all of them.

16.If even one of them sends a NO, ABORT the transaction and restart the transaction.

17.if one of them do not reply/ one server crashes COMMIT is sent

18.Read OK from Servers.

19.Write the response to the client.

**Backend Servers**

1.Structure called RECORDS is used for storing the account_number and account_balance.

2.It takes 3 command line arguments i.e servername, coordinator_port and Server_number

3.'Transaction_record' file will be created for each of the server

4.fopen(file_name,"w")- is used to read the file with write permission

5.**setsockopt(socketfd**, **SOL_SOCKET, SO_REUSEADDR**, (char *) &reuse, sizeof(reuse))- used to reuse the port

6.listening to connect to the clients

7.Reading the shared buffer for the commands from frontend server

8.copy the transaction to "transaction" array

9.Writting YES or NO to the coordinator.

10.reading commit/abort msg from coordinator

11.check if it is global commit msg or abort msg

12.parse transaction array and perform the task

13.If create , create an account id and return the account number

14.If QUERY , query for checking account balance and return account number

15.If UPDATE, update a record and return account number

16.If QUIT, send ready to commit msg to coordinator

17.If it is global abort dont save data into file

## CLIENT functionality

1.Socket structures are defined

2.Client accepts hostname and portnumber as the argument

3.Connects to the frontend sserver

4.Receives OK from the coordinator

5.Receives commands from the user

6.writing the command input taken by user to buffer

7.converting the command to uppercase if the command is in lowercase

by **using islower() and toupper()** functions.

8.writing to the coordinator

9.reading the final response from the coordinator

10.checking whether it is a OK msg from coordinator to close the connection.

11.closing the socket

TWO PHASE COMMIT PROTOCOL

**Commit request phase**

or **voting phase**

1. The coordinator sends a **query to commit** message to all cohorts and waits until it has received a reply from all cohorts.
2. The cohorts execute the transaction up to the point where they will be asked to commit. They each write an entry to their *undo log* and an entry to their *redo log*.
3. Each cohort replies with an **agreement** message (cohort votes **Yes** to commit), if the cohort's actions succeeded, or an **abort** message (cohort votes **No**, not to commit), if the cohort experiences a failure that will make it impossible to commit.

**Commit phase**

or **Completion phase**

**Success**[

**If the coordinator received an** agreement **message from *all* cohorts during the commit-request phase:**

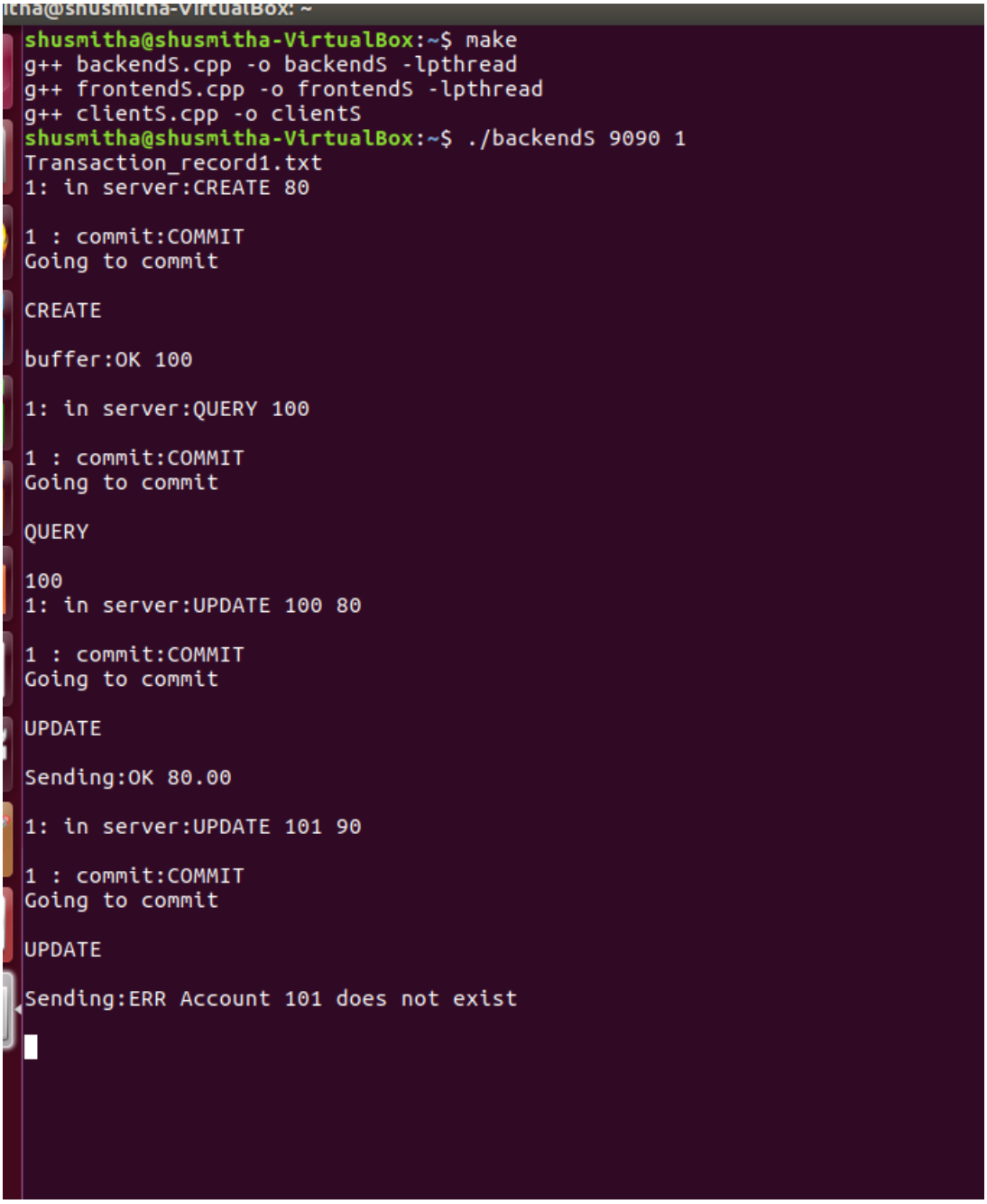
1. The coordinator sends a **commit** message to all the cohorts.
2. Each cohort completes the operation, and releases all the locks and resources held during the transaction.
3. Each cohort sends an **acknowledgment** to the coordinator.
4. The coordinator completes the transaction when all acknowledgments have been received.

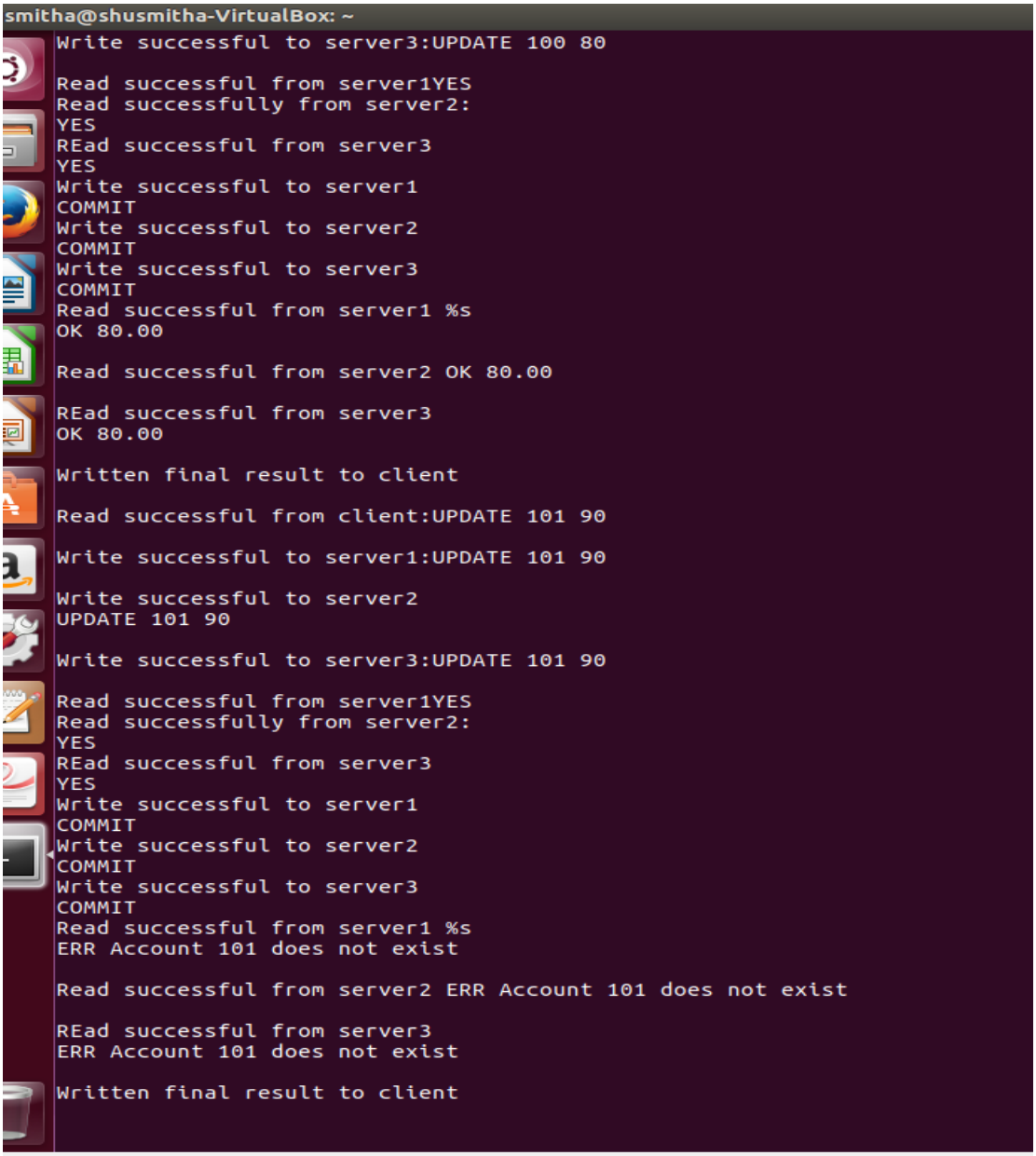
**Failure**

If *any* cohort votes **No** during the commit-request phase (or the coordinator's timeout **expires**):

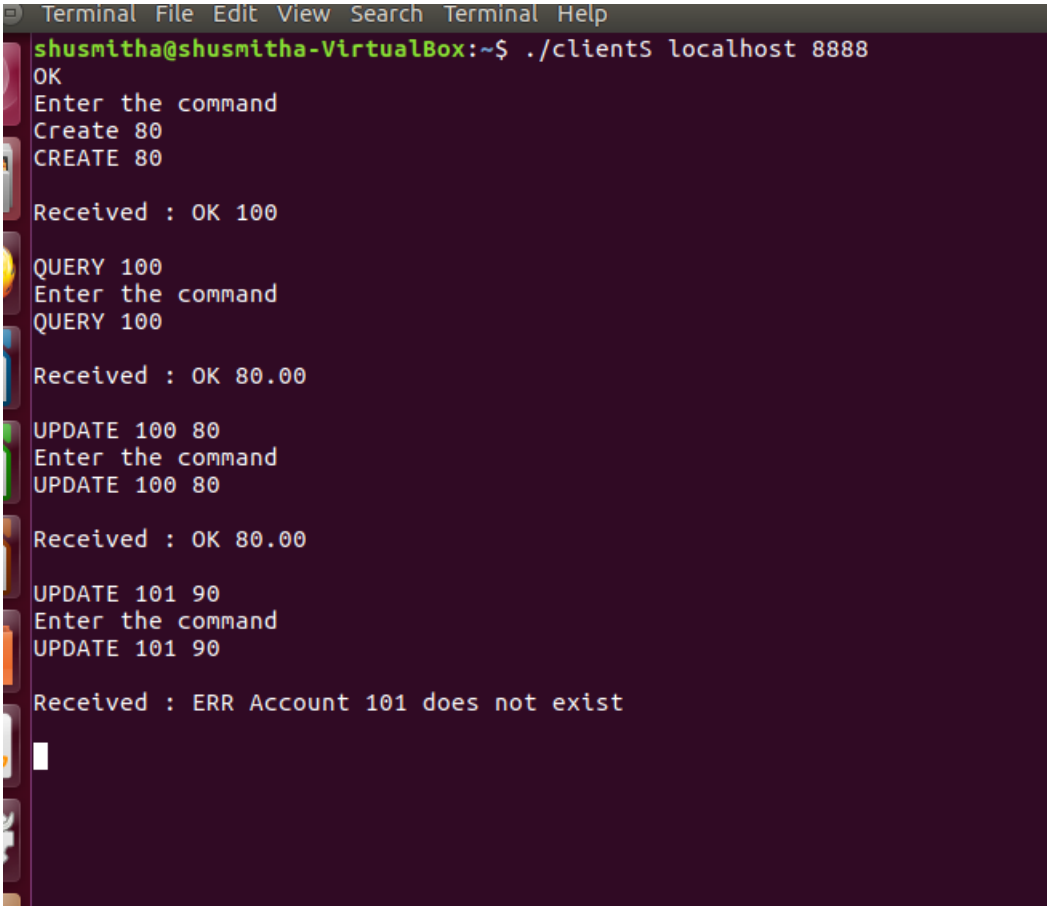
1. The coordinator sends a **rollback** message to all the cohorts.
2. Each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction.
3. Each cohort sends an **acknowledgement** to the coordinator.
4. The coordinator undoes the transaction when all acknowledgements have been received.

# OUTPUT :


shusmitha@shusmitha-VirtualBox:~$ make
g++ backendS.cpp -o backendS -lpthread
g++ frontendS.cpp -o frontendS -lpthread
g++ clientS.cpp -o clientS
shusmitha@shusmitha-VirtualBox:~$ ./backendS 9090 1
Transaction_record1.txt
1: in server:CREATE 80
1 : commit:COMMIT
Going to commit
CREATE
buffer:OK 100
1: in server:QUERY 100
1 : commit:COMMIT
Going to commit
QUERY
100
1: in server:UPDATE 100 80
1 : commit:COMMIT
Going to commit
UPDATE
Sending:OK 80.00
1: in server:UPDATE 101 90
1 : commit:COMMIT
Going to commit
UPDATE
Sending:ERR Account 101 does not exist




FRONTEND SERVER OUTPUT




OUTPUT OF CLIENT PROGRAM