```
/*
 *
 *  TEAM MEMBERS:
 *
 *  GAUTHAM REDDY KUNTA ID: 109596312
 *  NAFEES AHMED ABDUL  ID: 109595182
 *
 *
 *  CLIENT PSEUDO CODE
 *
 */
--------------------------------------------------------------------------
-----
/*
 Each client requests master for knowing the current head and tail of the
server chain they belong.
*/

my_address
master_address

var head initially null;
var tail initially null;



/*************** ALL REQUIRED EVENTS FROM MASTER ****************************/

event receive_UDP(“SetHead”,head_address) from master:
        head = head_address

event receive_UDP(“SetTail”,tail_address) from master:
        tail = tail_address

/*************** ALL REQUIRED EVENTS FROM Servers ****************************/
event receive_UDP(“response”,REPLY) from server_tail:
    responses := responses U {(server_tail, r)};


/*

 *  Function : init()

 *

 *  This method is called during the initial client setup.
```

```
 *  Each client obtains the head and tail from server
 */

 function init():
Begin:
        my_address = read from configuration file
        master_address = read from configuration file

        accountNo,bankName are input values for a given client
        send_UDP(“GetHead”,bankName, accountNo) to master
        send_UDP(“GetTail”,bankName, accountNo) to master
end:

/*

 *  Function : getBalance()

 *

 *  This method calls the head of the server to execute the corresponding
query.
 */




function getBalance(reqID, accountNo)
        reqId = generate unique ID for the request using client name , bank
name and unique sequence number.
            This sequence number should be unique and proper steps to be taken
such as using locks to
            preserve concurrent access across multiple threads(clients)
        repeat
            if servers = empty return ERROR(“unavailable”);
            send(“getBalance”, (reqID,accountNo)) to tail;

            wait up to T secs until Reply in responses;

            if ∃H (·, (reqId,Reply)) ∈ responses then
                return Reply or Display Reply to user;
end


/*

 *  Function : deposit()
```

```
 *

 *  This method calls the tail of the server to execute the corresponding
update
 *  query.
 */
function deposit(accountNo,amt) :
begin:
         reqId = generate unique ID for the request using client name , bank
name and unique sequence number.
            This sequence number should be unique and proper steps to be taken
such as using locks to
            preserve concurrent access across multiple threads(clients)
        repeat
            if head = empty return ERROR(“unavailable”);
            send(“deposit”, (reqID,accountNo,amt)) to head;
            wait up to T secs until Reply in responses;
            if ∃H (·, (reqId,Reply)) ∈ responses then
                return Reply or Display Reply to user;
end


/*

 *  Function : withdraw()

 *

 *  This method calls the head of the server to execute the corresponding
query.
 */

function withdraw(accountNo,amt) :
begin:
        reqId = generate unique ID for the request using client name , bank
name and unique sequence number.
            This sequence number should be unique and proper steps to be taken
such as using locks to
            preserve concurrent access across multiple threads(clients)
    repeat
        if head = empty return ERROR(“unavailable”);

            send(“withdraw”, (reqID,accountNo,amt)) to head;
            wait up to T secs until Reply in responses;
        if ∃H (·, (reqId,Reply)) ∈ responses then
                return Reply;
end
```

```
/*

 *  Function : transfer()

 *

 *  This method calls the head of the server to execute the corresponding
query.
 */

function transfer(accountNo,amt,destBankName, destAccNo) :
begin:
        reqId = generate unique ID for the request using client name , bank
name and unique sequence number.
            This sequence number should be unique and proper steps to be taken
such as using locks to
            preserve concurrent access across multiple threads(clients)
        repeat
             if head = empty return ERROR(“unavailable”);
               send(“transfer”, (reqID,accountNo,amt,destBankName, destAccNo))
to head;

            wait up to T secs until Reply in responses;
            if ∃H (·, (reqId,Reply)) ∈ responses then
                return Reply;
end
```

```
/*
 *
 *  TEAM MEMBERS:
 *
 *  GAUTHAM REDDY KUNTA ID: 109596312
 *  NAFEES AHMED ABDUL  ID: 109595182
 *
 *
 *  SERVER PSEUDO CODE
 *
 */
-----------------------------------------------------------------------
const my address “server address”
const master “master address”
enum Outcome { Processed, InconsistentWithHistory, InsufficientFunds }


class Reply {
  string reqID;
  Outcome outcome;
  float balance;
}

/* Account Class:
 * accountNo : Account number of a client in a bank
 * processedTrans : List of processed transactions for a particular account
 * currentTrans : List of current transactions which are processed but
 * not updated to client
 * amount : Current balance in the account
 */


class Account {
  string accountNo;
  List processedTrans;
  List currentTrans;
  float amount;
}



/*
 *  Account_List : List of all existing accounts that bank holds.
 *  Each account is of type class Account
 *  Server chain: contains chain/list of servers linked with a particular bank.
 *  isTail : Current Server is tail or not
 *  Responses : Response from master to handle transfer work flow
 */
```

```
var Account_List {A1, ..Ak}
var head
var tail
var predecessor
var successor
var isTail = false;
var Responses {}


/************** ALL REQUIRED EVENTS FROM MASTER ******************/

event receive_TCP(“Status”, message, new_server_address) from master:
    if message is “NEW TAIL”
        updateNewTail(null)
    if message is “NOT A TAIL”
        updateNewTail(new_server_address)


event receive_TCP (“getCurrentState”) from master
    get_CurrentState()


event receive_TCP (“updateState”, current_transactions) from master
    update_State(current_transactions);


event receive_TCP(“SetHead”, _HEAD) from master:
    head = _HEAD

event receive_TCP(“SetTail”, _TAIL) from master:
    tail = _TAIL

event receive_TCP(“SetPredecessor”, _PREDECESSOR) from master:
    predecessor = _PREDECESSOR

event receive_TCP(“SetSuccessor”,_SUCCESSOR) from master:
    successor = _SUCCESSOR


/************** ALL REQUIRED EVENTS FROM CLIENT **********************/

event receive_UDP (“getBalance”, reqId,accountNo) from client:
    getBalance(reqId,accountNo);

/* Receive from UDP port when receiving from client and
 * from TCP port when receiving from server
```

```
 */

event receive(“deposit”, reqID,accountNo,amt) from client/other server(incase
of transfer):
    deposit(reqID,reqID,accountNo,amt);

event receive_UDP(“withdraw”, reqID,accountNo,amt) from client:
    withdraw(reqID,reqID,accountNo,amt);

event receive_UDP(“transfer”, reqId, accountNo,amt,destBankName, destAccNo)
from client:
    transfer(reqId, accountNo,amt,destBankName, destAccNo);


/************** ALL REQUIRED EVENTS FROM OTHER SERVERS ******************/

event receive_TCP(“Confirm”, transaction, accountNo) from successor
    send_ack(transaction,accountNo)


event receive_TCP(“Sync”, client,Reply,accountNo,transaction) from predecessor
    Sync(client,Reply,accountNo,transaction)

event receive_TCP(“Sync”, states,accounts) from predecessor
    Sync(states,accounts)

event receive_TCP(“UpdateNewTailState”, Account_list) from previous_tail
    UpdateNewTailState(Account_list, previous_tail)

event receive_TCP(“response”, r) from other_bank
    Responses = Responses U {other_bank, r};

------------------------------------------------------------------------


/*
 *  Function : init()
 *
 *  This method is called when during initial server setup.
 *  Each server chain is obtained by the server from master at the
 *  time when its up and running.
 *
 *  1. Read the configuration and get its details
 *  2. Get the server chain from master.
 *  3. Create list for Account objects in the bank. As the request come
 *     we add to this list if it is not present.
 *  4. Create sockets for TCP and UDP communication with other servers,master
 *     and client.
```

```
 *  5. Create a new thread which takes care of pinging master about its
existence
 */

function init():
Begin:
    my address = read configuration file to get server address
    master = readConfigurationFile to get master address

    createTCPSocket() for server/master communication
    createUDOSocket() for client communication

    send_TCP(“GetHead”, my address) to master.
    send_TCP(“GetTail”, my address) to master.
    send_TCP(“GetPredecessor”, my address) to master.
    send_TCP(“GetSuccessor”, my address) to master.

    Account_List initially {}

    /* Extending Chain when new server wants to add itself into existing
chain*/
    send_TCP("Addtochain",my address,bankname) to master


    Thread thread;
    thread.start;
    if my address is tail
        isTail = true
End


/* Function run()
 * Thread's runnable method which sends its address to master for every T secs
 * to prove its existence
 */


function run()
Begin:
    send_TCP(“Ping_from_server”, my address) to master every T secs;
End


/* Function updateNewTail()
 * This method is called when either tail is failed
 * or new server is added to existing chain
 */
```

```
function updateNewTail(new_server_address)
Begin:
    if new_server_address is null
        previous_server = predecessor;
        for account in Account_List
            temp_trans = account.currentTrans
            account.processedTrans = account.processedTrans U
account.currentTrans;

            Reply = construct from account.currentTrans;
            account.currentTrans = empty;
            send_UDP(“response”, Reply) to client;
            send_TCP(“confirm”,temp_Trans, account.accountNo) to
previous_server;
    else
        send_TCP(“UpdateNewTailState”, Account_list, my address) to
new_server_address;
End


/* Function send_ack()
 *
 * This method sends acknowledgement to other servers in chain
 * when current request is accomplished(reply sent to client) so
 * that transaction can be added to
 * processedTrans and can be removed from currentTrans to have
 * consistency among the states along the replicas.
 */

function send_ack(transaction, accountNo):
Begin:
    previous_server = predecessor;

    Account current_account = getAccount(accountNo);

    current_account.processedTrans.add(transaction);
    current_account.currentTrans.remove(transaction)

    if previous_server is not null
        send_TCP(“confirm”, transaction, accountNo) to previous_server

End

/* Function updateNewTailState()
 *
 * This method is used to update states of newly created server
```

```
 *  in the chain so as to have consistency with
 * other servers in the chain
 *
 */

function updateNewTailState(old_Account_list, previous_tail)
Begin:
    Update the accounts with the snapshot of previous tail.
    Account_list = old_Account_list;
    for each account in Account_list
        for transaction in account.currentTrans
        {
            account.processedTrans.add(transaction);
            account.currentTrans.remove(transaction);
            Reply = construct from account.currentTrans;
            send_UDP(“response”, Reply) to client;
            send_TCP(“Confirm”, transaction, accountNo) to previous_tail;
        }
    send_TCP(“done”,my address,previous_tail) to master;
End


/* Function update_State()
 *
 *  This method is used to allot/update desired states to
 *  server(Successor of failed server) when there is failure of a server
 *  among internal servers of the chain
 */

function update_State(current_transactions)
Begin:
    for account in Account_List
        current_states = current_states U account.currentTrans

    states = current_states - current_transactions
    accounts = get corresponding accounts for states_to_update

    next = successor;
    send_TCP(“sync”,states,accounts) to next
End


/* Function get_CurrentState()
 *
 *  This method is used to get all the current states of server (items in
currentTrans
 *  list) which
```

```
 *  may have processed at each replica but not sent to client.
 */

function get_CurrentState()
Begin:
    current_states = {}
    for account in Account_List
        current_states = current_states U account.currentTrans

    send_TCP(“currentStateInfo”, current_states,my address) to master;
End

/* Function getBalance()
 *
 *  This method is gives the existing balance information of the account. This
is Query
 *  operation
 */

function getBalance(reqId,accountNo)
Begin:
    /* Check if the accountNo is present in Account_List If not present add it
to Account_List with initial amount = 0.*/

    Account current_account = getAccount(accountNo);

    Reply.reqId = reqId,
    Reply.balance = current_account.amount;
    Reply.outcome = Outcome.Processed

    send_UDP(“response”, Reply) to client;
End

/* Function getAccount()
 *
 * This method is gives the account information for the given accountNo . If
account not
 * present , it creates
 *  new account with initial balance as zero
 */

function Account getAccount(accountNo)
Begin:
    Account current_account;

    if accountNo in Account_List:
        current_account = Account_List(accountNo).
```

```
    else
        add current_account to Account_List with current_account.amount = 0
initially;

    return current_account
End

/* Function Validate()
 *
 *  This method performs validation
 *  Case 1 : Exactly same transaction has already been processed , simply
return the
 *  output(Don't process again)
 *  Case 2 : Different transaction with same request id , return as
Inconsistent with
 *  history
 *  Case 3 : This is success condition , go ahead processing the requests
 */


function int Validate(Account account, reqID,accountNo,amt,type)
Begin:
    foreach request in account.processedTrans:
    {
        if <reqID,type,accountNo,amt> matches request
            return 0;

        else if <reqID> matches request
            return -1
        else
            return 1;
    }
End

/* Function deposit()
 *
 *  This method performs deposit operation and let other replicas synchronizes
with this
 *  operation. This is one type of update operation
 */


function deposit (reqID,accountNo,amt)
Begin:
    Account current_account = getAccount(accountNo);
    result = Validate(current_account, reqID,accountNo,amt,”deposit”);
```

```
    Reply.reqId = ;

    if (result == 1)
    {
        current_account.amount = current_account.amount + amt;
        Reply.balance = current_account.amount;
        Reply.outcome = Outcome.Processed;
    }
    else if (result == -1)
    {
        Reply.balance = current_account.amount;
        Reply.outcome = Outcome. InconsistentWithHistory
    }
    else
    {
        Reply.balance = current_account.amount;
        Reply.outcome = Outcome.Processed;

    }

    transaction = <reqID,type,accountNo,amt>;
    current_account.currentTrans.add(transaction);
    next = successor;
    client = get client address form reqID
    send_TCP("sync", client, Reply,accountNo,transaction) to next;
End

/* Function withdraw()
 *
 *  This method performs withdraw operation and let other replicas synchronizes
with this
 *  operation. This is one type of update operation
 */

function withdraw (reqID,accountNo,amt)
Begin:

    Account current_account = getAccount(accountNo);

    result = Validate(current_account, reqID);
    Reply.reqId = reqId;
    if (result)
    {
        if (current_account.amount < amt)
            Reply.balance = current_account.amount;
            Reply.outcome = Outcome.InsufficientFunds;
        else
```

```
            current_account.amount = current_account.amount - amt;
            Reply.balance = current_account.amount;
            Reply.outcome = Outcome.Processed;
    }
    else if (result == -1)
    {
        Reply.balance = current_account.amount;
        Reply.outcome = Outcome.InconsistentWithHistory
    }

    transaction = <reqID,type,accountNo,amt>;
    current_account.currentTrans.add(transaction);
    client = get client address form reqID

    next = successor;
    send_TCP("sync", client, Reply,accountNo,transaction) to next
End


/* Function transfer()
 *
 *  This method performs transfer operation and let other replicas synchronizes
with this
 *  operation. This is one type of update operation . The head of server chain
related to
    source bank sends request by generating new request id to head of server
chain related
    to destination bank . Once this chain processes the request its tail sends
back reply
    to source head (not to client) . Then this request again propagates within
source bank
    chain and when it reaches tail , reply is sent back to client . The source
head waits
    until T time after it sends request to destination head . When this time is
elapsed
    and if it won't have any reply which means a probable failure we don't
serve the
    request , which means that client will be resending its request. The main
reason for
    this is to avoid blocking of server by waiting . We are ensuring that
change is
    reflected across all the destination and source bank servers.
 */

function transfer (reqId, accountNo,amt, destination_BankName,
destination_AccNo)
Begin:
```

```
    Account current_account = getAccount(accountNo);

    result = Validate(current_account, reqID);
    Reply.reqId = reqId;
    successful = true;
    if (result)
    {
        if (current_account.amount < amt) {
            Reply.balance = current_account.amount;
            Reply.outcome = Outcome.InsufficientFunds;
        }
        else {
            destination_Head = get the head from master for
destination_BankName
            request_ID = generate a new request id.s
            send_TCP("deposit",request_ID, destAccNo,amt) to destination_Head
            wait till response from destination_tail or T secs;
            if response from destination_BankName
                current_account.amount = current_account.amount - amt;
                Reply.balance = current_account.amount;
                Reply.outcome = Outcome.Processed;

            else
                    success = false;
        }
    }
    else if (result == -1){
        Reply.balance = current_account.amount;
        Reply.outcome = Outcome.InconsistentWithHistory
    }

    if (success) {
        transaction = <reqID,type,accountNo,amt>;
        current_account.currentTrans.add(transaction);
        next = successor;
        client = get client address form reqID
        send_TCP("sync", client, Reply,accountNo,transaction) to next
    }
End

/* Function sync()
 *
 *  This method is used to synchronize other replicas with the consistent
states and takes
 *  care of replying back to client if server is tail , otherwise recursive
 *  synchronization  takes place across servers of the chain
 */
```

```
function sync(client,Reply,accountNo,transaction)
Begin:

    Account current_account = getAccount(accountNo);
    current_account.amount = Reply.balance;


    current_account.currentTrans.add(transaction);

    next_server = successor;

    if next_server is not null then
        send_TCP(“sync”, (client,Reply,accountNo) to next_server
    else
        previous_server = predecessor;
        if (isTail) {

            current_account.processedTrans.add(transaction);
            current_account.currentTrans.remove(transaction)

            send_UDP(“response”, Reply) to client
            send_TCP(“Confirm”, transaction, accountNo) to previous_server
        }

End

/* Function sync()
 *
 *  A different type of sync implementation which essentially does the same
thing
 *  as above method but used during failure cases of server
 */


function sync(states,accounts)
Begin:

    for account no in accounts using states information particularly accountNo
{
        Get balance from account
        if account no in Account_list:
            update existing balance with new balance
    }

    current_account.currentTrans.add(states);
    next_server = successor;
```

```
    if next_server is not null then {
        accounts = get corresponding accounts for states
        send_TCP(“sync”, (states, accounts))to next_server
    }
    else {
        previous_server = predecessor;

        Reply = construct from states
        accountNo = get account No from corresponding processed account;
        send_UDP(“response”, Reply) to client;
        send_TCP(“Confirm”, states, accountNo) to previous_server;
    }
End
```

```
/*
 *
 *  TEAM MEMBERS:
 *
 *  GAUTHAM REDDY KUNTA ID: 109596312
 *  NAFEES AHMED ABDUL  ID: 109595182
 *
 *
 *  MASTER PSEUDO CODE
 *
 */
-------------------------------------------------------------------------------
-----

const master “master address”
const servers_chains = 3
var current_state initially null
var server_chain_bank = {}
var Server_bank_active list = {}
var bank1_clients_list = {}

-----------------------------------------------------------------

/************** ALL REQUIRED EVENTS FROM SERVER
**********************************/
/*
 *  Receives message from each server of the chain for every T secs
 *
 */
event receive_TCP (“Ping_from_server”, server) from server:

    ## validate which chain it belongs to and we can keep track of live
       servers in that chain.

    if server belong to server_chain list:
        Server_bank1_active[server] = 1;


event receive_TCP (“GetHead”, server) from server:

    if server belong to serverChain list:
        send_TCP(“SetHead”, serverChain->head)

event receive_TCP (“GetTail”, server) from server:
    if server belong to serverChain list:
        send_TCP(“SetTail”, serverChain->tail)
```

```
event receive_TCP (“GetPredecessor”, server) from server:
    if server belong to serverChain list:
        send_TCP(“SetPredecessor”, serverChain[server]->predecessor)

event receive_TCP (“GetSuccessor”, server) from server:
    if server belong to serverChain list:
        send_TCP(“SetSuccessor”, serverChain[server]->successor)

event receive_TCP (“Done”, server_previous,server_tail) from server:

    if server_previous belong to serverChain list:
        update serverChain list with server_tail
        send_TCP(“SetTail”, server_tail) for all servers in the given Server
Chain
        send_UDP(“New Tail”, server_tail) to all clients in bank_client_list


event receive_TCP (“currentStateInfo”, currentState, server) from server:
     current_state = currentState;

event receive_TCP (“AddToChain”, server, bankName) from server:
     addNewServertoChain(server, bankName);

/************** ALL REQUIRED EVENTS FROM CLIENT
**********************************/


event receive_UDP (“GetHead”, BankName ,clientID/acc No) from client:

    if BankName is banks
        bank_clients_list = bank_clients_list U {clientID/accNo}
        send_UDP(“SetHead”, serverChain->head)to client;
    else
        send_UDP(“SetHead”, null)to client;


event receive_UDP (“GetTail”, BankName ,clientID/acc No) from client:
    if BankName is banks
        bank_clients_list = bank_clients_list U {clientID/accNo}
        send_UDP(“SetTail”, serverChain->head)to client;
    else
        send_UDP(“SetTail”, null)to client;

_____________________________________________________________________________
_______
/*
```

```
 *  Function : init()
 *
 *  1.Read Configuration file to get required details
 *  2.Create two sockets both TCP and UDP for communications with server and
client
 *    respectively
 *  3.Initialize server chain bank and their status (active or not)
 *  4.Create a thread to checks existence of server
 */


function init():
Begin:
    my address = read configuration file to get master address
    createTCPSocket() for server communication
    createUDPSocket() for client communication

    server_chain_bank = Get from configuration file for a bank
    /* Similarly we can get configurations for other bank chains and active
list*/

    Server_bank_active, initialise list to zero

    Thread isServerAlive;
    isServerAlive.start();

End

/*
 * This thread is always validating the server chains for failures
 */

function run()
Begin
    for every T secs
        check if Server_bank_active is all 1's.
        if yes
            Server_bank_active = all 0's
        else
            get the ID's of the server whose value is 0.
            updateServerChain(ID_List,server_chain);

/*  Similarly we do for other bank chains*/

End

/*
```

```
 *  Function : updateServerChain()
 *
 *  1.Method used to handle different failure conditions
 *  2.If failure at head , then make it's successor as head and update
 *    about new head to respective servers and clients and remove failed head
from server
 *     list
 *  3.If failure at tail remove tail from server list and make it's predecessor
as new
 *     tail
 *  4.Handles internal server failure cases appropriately.
 */

function updateServerChain(serverId, serverChain):
Begin:
    if serverId is serverChain->head then
        newHead = serverChain[serverId]->successor;

        serverChain->head = newHead
        update serverChain list with new head and remove the old head
        send_TCP("SetHead", newHead) for all servers in the given updated
Server Chain
        send_UDP("New Head", newHead) to all clients in bank_client_list of
serverChain


    else serverId is serverChain->tail then
        newTail = serverChain[serverId]->predecessor;

        serverChain->tail = newTail
        update serverChain list with new tail and remove the old tail
        send_TCP("SetTail", newTail) for all servers in the given updated
Server Chain
        send_TCP("NEW TAIL", newTail) to newTail
        send_UDP("New Tail", newTail) to all clients in bank_client_list
    else
        update serverChain list by removing the serverId
        send_TCP("getCurrentState") to serverId->successor
        wait for response of current_state
        send_TCP("updateState", currentState) to serverId->predecessor

End

/*
 *  Function : addNewServertoChain()
 *
 *  Method used while extending existing chain of servers , that is appending
```

```
new servers
 */

function addNewServertoChain(server, bankName):
Begin:
    serverChain = Get the serverChain for bankName;
    tail  = serverChain->tail
    send_TCP(“Status”, “NOT A TAIL”, server) to tail;
End
```