Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 1 & Section 3 – Pseudocode

```
//Pseudocode for ChainReplicationClient

const T = retransmission delay;
var servers initially {p1,..., pn};
var replies initially NULL;
var head = NULL;
var tail = NULL;
var Chains; (from Config file)
var chain;
var myBankName; initialize to 'this' client's bankName; (from Config file)
Static var seqNo = 0;//This is a global variable instance and its values are
prevalent in all threads of clients.

//This method is an abstraction for all communication between clients and servers.
This need not be a strictly reliable protocol. Hence UDP is employed for
communications.
function send(...)//variable parameters
      establish UDP connection between client and server and send the data
end

//Event to receive information about the head of chains
event receive("updateHead", (chain, P)) :
      Chains[chain].head = P
      this.chain = chain;
end

//Event to receive information about the tail of chains
event receive("updateTail", (chain, P)) :
      Chains[chain].tail = P
      this.chain = chain;
end

//Event which receives responses for all the requests from servers
event receive(“Reply”, r) from p :
      replies := replies ∪ {(p,r)};
end

//Returns the head of the chain
function head(servers, chain)
      if(servers is empty)
            return NULL;
      return P; the first process in ‘servers’ that belong to ‘chain’ such that P >
      all other processes in servers in the chain.
end

//Returns the tail of the chain
function tail(servers, chain)
      if(servers is empty)
            return NULL;
      return P; the last process in ‘servers’ that belong to ‘chain’ such that P <
      all other processes in servers
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

```
//Request from client to retrieve the balance amount in the account
function getBalance(reqId, accountNum)
	if server = ⊥ then
	reqId := getRequestId();
	repeat:
		if servers = NULL then return ERROR(“unavailable”);
		this.tail := min(servers);

		if(this.tail = NULL)
			return NULL;

		send(“getBalance”, reqID) to tail;

		wait until (reqId,·,·) ∈ repliesVtail !∈ servers ;

		if ∃r (reqID,·,·) ∈ replies then
			if r contains (Processed)
				return r;
			else
				//do nothing so that the server is requested again. (do not
				call return)
end

//Request from client to deposit money into the account
function deposit(reqID, accountNum, amount):
	reqID := getRequestId();
	repeat:
		if servers = NULL then return ERROR(“unavailable”);

		this.head := max(servers);

		if(this.head = NULL)
			return NULL;

		send(“deposit”, (reqID, accountNum, amount)) to this.head;

		wait up to T seconds until (reqID,·,·) ∈ replies;

		if ∃r (reqID,·,·) ∈ replies then
			if r contains (Processed)
				return r;
			if r contains (InconsistentWithHistory)
				//do nothing so that the server is requested again. (do not
				call return)
end

//Request from client to withdraw money from the account
function withdraw(reqId, accountNum, amount):
	reqID := getRequestId();
	repeat:

	if servers = NULL then return ERROR(“unavailable”);

	this.head := max(servers);
		if(this.head = NULL)
```

```
                    return NULL;

        send("withdraw", (reqID, accountNum, amount)) to this.head;

        wait up to T seconds until (reqID,·,·) ∈ replies;

        if ∃r (reqID,·,·) ∈ replies then
              if r contains (Processed)
                    return r;
              if r contains (InconsistentWithHistory ∨ InsufficientFunds)
                    //do nothing so that the server is requested again. (do not call
                    return)
end

//Request from client to initiate a wire transfer between the bank this client is
currently representing to another bank
function transfer(reqID, accountNum, amount, destBank, destAccount)
        reqId := getRequestId();
        repeat:
              if servers = NULL then return ERROR("unavailable");

              this.head := Chains.bankName[accountNum].chain.head;
              if(this.head = NULL)
                    return NULL;

              //This is the starting point for wire transfer processing in origin bank
              send("receiveTransferWithdraw", (reqID, accountNum, amount)) to
              this.head;

              //client upon receiving reply from servers or upon not getting reply at
              all, checks if the request is processed. If not, it reinitiates the
              request.
              wait up to T seconds until (reqID,·,·) ∈ replies;

              if ∃r (reqID,·,·) ∈ replies then
                    if r contains (Processed)
                          return r;
                    if r contains (InconsistentWithHistory ∨ InsufficientFunds)
                          //do nothing so that the server is requested again. (do not
                          call return)
end

//Returns a unique id for each request made by the client to any servers. The
sequence number used is a global variable and doesn't change with each client. Hence
if this method is used for all calls from clients to all other servers, then the
requestId will be unique across the system.
function getRequestId()
        seqNo++;

        return this.myBankName.toString() + this.clientNumber.toString() +
        seqNo.toString();//concatenate all the individual strings
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

```
//Initializatio method for clients. This method spawns threads, each representing a
client with data to perform the pre-determined probabilistic requests. Probability of
request information will be retrieved from the config file.
function clientInit()
      Read configuration file to get the details of client instances required
      Spawn threads for each client and supply the details fetched
end

//Event to receive notification about the failure of head for a particular chain
event receive("headFailed", (chain, P))
      Stop processing any requests on this chain.
      OR retransmit the request after the timeout. //Yet to be decided
end

//Event to receive notification about the failure of tail for a particular chain
event receive("tailFailed", (chain, P))
      Stop processing any requests on this chain.
      OR retransmit the request after the timeout. //Yet to be decided
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 2 – Pseudocode

THE COMPLETE DESCRIPTION OF THE PSEDUCODE

THE CLIENT:

The client always initiates the requests for deposit, withdraw and the transfer operations.

1. It receives the updated head and tail from every chain by the master through receive "updateHead" and "updateTail" events.
2. It receives reply from the tail when a query or an update is processed.
3. The function head( ) returns the return first process P in servers such that P > all other processes in servers as head.
4. The function tail( ) return last process P in servers such that P < all other processes in servers.
5. The function getBalance( ) returns the balance in the account at any point of time to the client.
6. The function deposit ( ) and withdraw( ) and transfer( ) is carried out to put in a valid amount to the account. This update operation is carried by head and notified to the client.
7. The function getRequestId( ) returns the id in the form of bankName.clientNumber.SequenceNo.
8. The function clientInit ( ) spawns threads for each client and supply the details fetched.
9. The events headFailed( ) and tailFailed( ) stop processing any requests on this chain.

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 1 & Section 3 – Pseudocode

```
//Pseudocode for ChainReplicationServer

//Server, in this case is a replicated storage system of the bank.
const me = “my address”;
var State initially ⊥;
var servers initially {p1,..., pn};
var ProcessStates initially ⊥;
var SentList initially ⊥;
var predecessor = NULL;
var successor = NULL;
var chainServices = NULL; //This holds information about head and tails of all the
chains including oneself.

//This event is invoked by client while initiating a transfer request. This procedure
will redirect the request to the head of the chain, so that the processing can be
done by every server in chain in a sequential manner.
event receive("receiveTransferWithdraw", (reqId, accountNum, amount, destBank,
destAccount))
      send("processTransferWithdraw" (reqId, accountNum, amount, destBank,
      destAccount)) to chainServices[me].head;
end

//This procedure will only perform the balance withdrawal of source bank during a
transfer request. After withdrawal this request is propagated down the chain so that
all the living servers initiate the same withdrawal request. (to avoid balance
disappearing)
function processTransferWithdraw(reqId, accountNum, amount, destBank, destAccount)
      if(successor != NULL)

            send (“withdraw”, (reqID, accountNum, amount)) to 'me';
            //Remember that send within servers acts over TCP/IP. All the time-out
            //functionality is assumed to be encapsulated in the send method.

            tcpResponse = send (“receiveTransferDeposit”, (reqId, accountNum,
            amount, destBank, destAccount)) to

            chainServices.bankName[destBank].chain.head;

            send ("processTransferWithdraw", (reqId, accountNum, amount, destBank,
            destAccount)) to successor;
end

//Event to receive acknowledgement from the successors of this server. This indicates
that the request from clientNumber with reqId has been notified of the result of the
request by the tail of the chain.
event receive("ack", (clientNumber, reqId))
      if (predecessor != NULL)
      send("ack", (clientNumber, reqId)) to predecessor;
end

//Event to receive the trigger to deposit the amount in the destination bank in case
of transfer requests. This event will be triggered by each server of the origin bank
after the withdraw part of the transfers are completed by the origin banks.
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

```
event receive("receiveTransferDeposit", (reqId, accountNum, amount, destBank,
destAccount))
      send("processTransferDeposit" (reqId, accountNum, amount, destBank,
      destAccount)) to chainServices[me].head;
end

//This is the action taken to perform the deposit part of transfer by the server upon
receiving notification from the previous server.
function processTransferDeposit (reqId, accountNum, amount, destBank, destAccount)
      if(successor != NULL)
            send (“deposit”, (reqID, accountNum, amount)) to 'me';

      send ("processTransferWithdraw", (reqId, accountNum, amount, destBank,
      destAccount)) to successor;
end

//This event notifies each server about the head and tail of all the servers
including oneself. This enables each server to be a client itself, in case of
transfer requests.
event receive("receiveChainInformation",(chain, head, tail))
      chainServices[chain].head = head;
      chainServices[chain].tail = tail;
end

//This method is an abstraction for all the communication between servers and other
servers including master. In order for chain replication protocol to work, this has
to be reliable. Hence employing TCP/IP for communication between the parties.
function send(...)//variable parameters
      establish TCP/IP connection between server and other servers and send the data
end

//Notification for server to realize its predecessor. This helps in identifying the
server’s responsibility.
event receive("updatePredecessor", (P)):
      predecessor = P
end

//Notification for server to realize its successor. This helps in identifying the
server’s responsibility.
event receive("updateSuccessor", (P)) :
      successor = P
end

//Procedure to initiate a deposit request
event receive(“deposit”, (reqID, accountNum, amount)) from client :
      ∀p ∈ servers : p > me ⇒ servers := servers\{p};
      if predecessor = NULL; //perform this action only if this is the head
            if (reqId, ·, ·) !∈ ProcessStates then
                  //Process the request and calculate the new state and update the
                  State variable.

                  balance = balance + amount;

                  //Update state to contain 'Processed'. r = <reqId, Processed,
                  balance>
```

```
                  State := State :: (reqID, accountNum, amount);

            else if (reqId, accountNum, amount) ∈ ProcessStates
                  //Different transaction with same requestId
                  //Update state to contain 'InconsistentWithHistory'. r = <reqId,
                  InconsistentWithHistory, balance>;
                  State := State :: (reqID, accountNum, amount);
            else
                  //Same transaction already been processed for this account
                  //Update state to contain 'Processed'. r = <reqId, Processed,
                  balance>;
                  State := State :: (reqID, accountNum, amount);

            sync(State, clientNumber , reqId);
end

//Procedure to initiate a withdraw request
event receive("withdraw", (reqID, accountNum, amount)) from client :
      ∀p ∈ servers : p > me ⇒ servers := servers\{p};
      if predecessor = NULL; //perform this action only if this is the head
            if (reqId, ·, ·) !∈ ProcessStates then
                  //Process the request and calculate the new state and update the
                  State variable.
                  if(balance >= amount)
                        balance = balance - amount;

                  else
                  //Update state to contain 'InsufficientFunds'. r = <reqId,
                  InsufficientFunds, balance>
                  //Update state to contain 'Processed'. r = <reqId, Processed,
                  balance>
                  State := State :: (reqID, accountNum, amount);

            else if (reqId, accountNum, amount) ∈ ProcessStates
                  //Different transaction with same requestId
                  //Update state to contain 'InconsistentWithHistory'. r = <reqId,
                  InconsistentWithHistory, balance>;
                  State := State :: (reqID, accountNum, amount);

            else
                  //Same transaction already been processed for this account
                  //Update state to contain 'Processed'. r = <reqId, Processed,
                  balance>;
                  State := State :: (reqID, accountNum, amount);

            sync(State, clientNumber , reqId);
end

//Notification for the server from it predecessors to update the state and forward
the request to successors of the chain until tail is reached
event receive("sync", (newState, clientNumber, reqId)) from prev :
      ∀p ∈ servers : me < p < prev ⇒ servers := servers\{p};
      if prev = predecessor(servers, me) then
            State := newState;
            sync(State, clientNumber, reqId);
```

```
end


//Procedure to update the state of the server and forward the transaction to other
servers in the chain. If the current server is a tail, it stops forwarding the
requests to other servers and informs the client about the result of the transaction.
function sync(State, clientNumber, reqId) :
      next = successor(servers, me);
      prev = predecessor(servers, me);
      if next != ⊥ then
            //Update the SentList to manage acknowledgements and to propagate the
            State in case of internal server failure.
            SentList := SentList . State;

            send(“sync”, (State, clientNumber, reqId)) to next;

      else
            //Outcome has been decided by 'head' and propagated here.
            //Sending the result to client.
            send(“Reply”, r=<reqId, outcome, balance>) to clientNumber;

            //Initiate acknowledgments
            sendAck("ack", (clientNumber, reqId)) to prev;
end

//Notification to servers from the master informing about the server failure in the
system.
event receive("failure", (chain,P))
      //Remove the server from the list
      servers := servers\{p};

//Initialize method for all servers. Here server creates a thread to inform master
about the liveness of the thread. If server crashes, this periodic notifications to
master stops and master will assume that the server is crashed after a particular
amount of time
function ServerInit()
      Spawn a server local thread to send an event to master's thread
      ServerHeartBeatTimeout units.// For example: 1 second
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 2 – Pseudocode

THE COMPLETE DESCRIPTION OF THE PSEDUCODE

THE SERVER:

The servers of a chain denote the different branches of a bank in the same chain. It evolves the following functionalities:

1. The event receive – "receiveTransferWithdraw" is received by the server from a client. The client sends the requested (which is an unique ID maintained for the transaction), acountnumber and the amount to be transferred(to be deposited or withdrawn), the destination bank name and account. As soon as the server of a chain gets the receive event, it sends the "processTransferWithdraw" message to the head with all the just mentioned parameters of its chain because the head is the starting point for all the updates computing. Here the processing can be done by every server in chain in a sequential manner. The chainServices[me].head indicates that every server is informed through this about the head.
2. The procedure processTransferWithdraw( ) will only perform the balance withdrawal of source bank during a transfer request.
3. After withdrawal this request is propagated down the chain so that all the living servers initiate the same withdrawal request. (To avoid balance disappearing). This will initiate a sending of the "withdraw" message to the server to which the transfer request was initially addressed. Since the send method acts over the TCP/IP connectin, the timeouts are encapsulated in the send ( ) method.
4. In a similar way, the event receive – "receiveTransferDeposit" is received by the server from a client. The client sends the requested (which is an unique ID maintained for the transaction), acountNumber and the amount to be transferred(to be deposited or withdrawn), the destination bank name and account. As soon as the server of a chain gets the receive event, it sends the "processTransferDeposit" message to the head with all the just mentioned parameters of its chain because the head is the centre for all the updates computing. Here the processing can be done by every server in chain in a sequential manner. The chainServices[me].head indicates that every server is informed through this about the head.
5. The procedure processTransferDeposit( ) will only perform the balance withdrawal of source bank during a transfer request.
6. After deposit, this request is propagated down the chain so that all the living servers initiate the same deposit request. (To avoid balance disappearing). This will initiate a sending of the "deposit" message to the server to which the transfer request was initially addressed. Since the send method acts over the TCP/IP connection, the timeouts are encompassed in the send ( ) method.

7. The computation is propagated down the chain to the tail. The tail is responsible for the sending of the response or reply messages to the client. This it does by send “ack” message to the client that initiated the request, identified by the request ID.
8. The event receive termed the “receiveChainInformation" is responsible for notifying the servers of the chains regarding the head and tail information which is to be maintained at the servers of all chains.
9. The function send (...) is responsible for establishing TCP/IP connection between server and other servers to send the data.
10. In case of a failure event, the master intervenes and finds – (a)the new head(in case if a head failure), (b)new tail(in case if a tail failure) , (c)new predecessor and successor in case of an internal node failure. And this updated events are notified to the server by “updateSuccessor” and “updatePredecessor” events/
11. The event receive deposit – updates the balance in case of a successful deposit. If different transaction with same request Id are requested, then updates the state to “InconsistentWithHistory”. And if a transaction is already processed, then returns the state. Sync is used to communicate to the next server about the state of the transactions performed.
12. The event receive withdraw– updates the balance in case of a successful deposit. If different transaction with same request Id are requested, then updates the state to “InconsistentWithHistory”. And if a transaction is already processed, then returns the state. Sync is used to communicate to the next server about the state of the transactions performed.
13. The function sync( ) - Updates the SentList to manage acknowledgements and to propagate the State in case of internal server failure. Then the result of the operations are sent to the client.
14. Function ServerInit( ) - Spawn a server local thread to send an event to master's thread ServerHeartBeatTimeout units - For example: 1 second.

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 1 & Section 3 – Pseudocode

```
//Pseudocode for ChainReplicationMaster

const me = "my address";
var chainLengths; (From Config file)
var BankNames; (From Config file)
var Clients; (From Config file)
var serverStartupDelay (From Config file)
var Chains;
var Servers;

//This event is a notification for any Server failures in the system. This event will
be triggered if Master fails to get a 'heart beat' event from each of the Server for
a specified amount of time.
event failure(P):
	servers := servers\{P}

	//broadcasts the failure event to all other servers
	foreach: chain in Chains
		foreach: server in Servers
			send("failure", (chain,P));
		endfor
	endfor

	//if the failed server is a head or tail of any chain, inform the Clients
	foreach: chain in Chains
		if (P = chain.head OR P = chain.tail)
			foreach: client in Clients
				send("headFailed", (chain,P)) to client;
				send("tailFailed", (chain,P)) to client;

	//Updates the head and tail of each client
	if P belongs_to Chain in Chains
		S- = predecessor(P)
		S+ = successor(P)
		if S- is NULL
			send("updateHead", P) to Clients of the bank to which this Chain
			belongs to;
		if S+ is NULL
			send("updateTail", P) to Clients of the bank to which this Chain
			belongs to;
		if S- != NULL AND S+ != NULL
			sequenceNumber = send("newRoleSuccessor") to S+;
			send("newRolePredecessor" (S+, sequenceNumber)) to S-;
end

//This function is an abstraction for the communication between Master and clients or
servers. This is a reliable communication. Hence TCP/IP protocol will be employed for
communication.
function send(...)//variable parameters
	establish TCP/IP connection between master and server/client and send the data
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

```
//Initialize method for Master. This is invoked only once and only when the master
server is booting up. Creates proceses (servers) based on information provided in
configuration file. Creates threads to listen to the heart beat messages from the
servers. Informs the clients about the head and tail of corresponding chains.
function MasterInit()
	foreach: bankName in BankNames
		Chain[j++] = create a chain and add it to Chains

		wait until 'serverStartupDelay' is expired;

		Servers[i++] = create as many processes required by the chain of this
		bank
	endfor

	//threads and server listeners
	foreach: server in Servers
		Spawn threads to listen to servers' heart beat for every
		‘ServerAliveTimeout’ units.// For example: 5 seconds

		register a listener from master which receives events from servers

		Wait until each thread for ServerAliveTimeout
			if event not received, then
				send(“failure”, server) to me(master);
	endfor

	//informing head and tail to clients
	foreach: bank in Banks
			chain = chain of this bank

			send(“updateHead”, findHead(chain)) to corresponding each client
			of bank;

			send(“updateTail”, findTail(chain)) to corresponding each client
			of bank;
	endfor

	//creating communication links between master and all other servers
	foreach: chain in Chains
		foreach: server in Servers
			establich TCP/IP connection between server and me (master)
		endfor
	endfor

	foreach: chain in Chains
		foreach: server in chain.Servers

			//Communicate to each server the head and tails of every other
			chain. This is needed to invoke services of other chains from
			servers of a particular chain. Like deposit part of transfer
			operation.

			foreach: chain in Chains
				send("receiveChainInformation", (chain, chain.head,
				chain.tail)) to server;
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

```
                        endfor

                endfor
        endfor
end

//Finds the head of the chain. This leverages on the fact that the process number of
head is bigger than all other processes in the chain.
function findHead(chain)
        head = NULL;

        foreach:server S in Chain.Servers
                if (S > S+)
                        head = S;
        endfor

        return head;
end

//Finds the tail of the chain. This leverages on the fact that the process numbr of
tail is smaller than all other processes in the chain.
function findTail(chain)
        tail = NULL;
        foreach:server S in Chain.Servers
                if (S < S+)
                        tail = S;
        endfor
        return tail;
end

//Finds the server next to the current server in the logical ordering of servers in
the chain
function successor(Servers, P)
        P+ = first process in Servers such that P+ < P
        if P+ is NULL
                return NULL;
        else
                return P+;
end

//Finds the server previous to the current server in the logical ordering of servers
in the chain
function predecessor(Servers, P)
        P- = first process in Servers such that P- > P
        if P- is NULL
                return NULL;
        else
                return P-;
end
```

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Assignment Section 2 – Pseudocode

THE COMPLETE DESCRIPTION OF THE PSEDUCODE

THE MASTER:

The primary activity of the master is the failure detection. There are three aspects of failure detection –

1. Failure of Head: Assuming that there is a server process P belonging to the class of servers, and if P becomes unavailable, the master intervenes and updates the event. The master checks if there is a predecessor S- for the server P, such that S- > P (P is the head). Here S- is NULL and thus the server after P gets promoted as the head of the chain. This is notified to all the clients, as sending information about the updated head is the part of the work designated to the master.
2. Failure of the Tail: Assuming that there is a server process P belonging to the class of servers, and if P becomes unavailable, the master intervenes and updates the event. The master checks if there is a successor S+ for the server P, such that S+ < P (P is the tail). Here S+ is NULL and thus the server just before P in the chain gets promoted as the new tail of the chain. This is notified to all the clients, as sending information about the updated tail is the second part of the work designated to the master.
3. Failure of any intermediate/internal server: If the server P is an internal server, it should possess servers S- and S+ as its neighbors. The master attempts to check if there are successors and predecessors for P and then sends a "newRoleSuccessor" message to the server S+ as part of the sequence number indicating that there is no longer server P and it has to resume its new position. And then S+ acknowledges this and sends a "newRolePredcessor" message to the server S- along with the sequence number.

The master broadcasts the event if failure to all the other servers in all the chains: The variable Chains contains the number of chains (equivalent to the number of banks present in the system) and the variable Servers contains the number of servers in each of individual chain. So in the event of a failure, all servers of all chains are made aware of this by sending a "failure" message. In the case of a head or tail server failure, all the clients (the variable Clients contain all the clients present in the system) by a "headFailed" and "tailFailed" notification messages.

The master is responsible for the following other functions:

1. The function send( ) essentially establishes an effective TCP/IP communication between the master and the servers and all clients to send the data.

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

2. The MasterInit( ) method: This method establishes the following functions:
    1. It creates a chain of servers for every branch and adds it to the variable Chains implemented as an array. We do initiate a parameter "serverStartupDelay" and wait until the server is up for the transfer operations. It creates as many processes as required by each bank in its chain by incrementing the array variable Servers[ ] for all chains.
    2. For every server process in Servers, the master –(a) Spawns threads to listen to servers' heart beat for every ServerAliveTimeout units. For example: 5 seconds (b) registers a listener from master which receives events from servers
    3. For every bank listed in the Bank variable, we initialize the chain to the chain of that bank, master finds the new Head and tail by the findHead(chain) and findTail(chain) functions. And the results are transmitted to each client of the bank by "updateHead" and "updateTail" messages.
    4. For each of the chain in Chains and for each of the servers in every chain, the master establishes a TCP/IP communication between each server and itself(master is dnoted as "me" by taking its own address).
    5. Communicate to each server the head and tails of every other chain. This is needed to invoke services of other chains from servers of a particular chain – for the deposit part of the transfer operation. This is done to notify all servers in all chains. And every chain in the system is made aware of the head and tail of every other chain by sending the "receiveChainInformation" to every server.

- The function findHead( ) checks for a server S > S+ in the chain and returns S as the head of the chain(assuming there are more than one servers in a chain).
- The function findTail( ) checks for a server S < S+ in the chain and returns S as the tail of the chain.
- The function successor( ) has 2 parameters – Servers and P, where P+ is the first process in Servers such that P+ < P. If P+ exists, it is returned as the successor of P.
- The function predecessor( ) has 2 parameters – Servers and P, where P- is the first process in Servers such that P- > P. If P- exists, it is returned as the successor of P.

Thus the master performs its primary functions, establishes communication between all the servers and clients and coordinates the chains in the system.

Supritha Mundaragi <smundaragi@cs.stonybrook.edu> (SBU_ID: 109797498)
Vishal Nayak <vnayak@cs.stonybrook.edu> (SBU_ID: 109892702)

Section 2 – Justification for the design

WHY IT WORKS – PROOF OF CORRECTNESS

The transfer operation can be verified for any transfer from the server of a chain of any bank to the server of the chain of the other bank to either deposit or withdraw a specific amount. The design so far works for the transfer operation. Suppose we assume that no response is received by the client after the operation is successfully performed. But the pseudo code specifies that the client is notified of the head and tail of all chains and that the servers know the head and tails of all banks. Thus any server forwards the transfer request (if it is not the head of the chain). The head processes the request, and sends the result to each of the internal servers in the chain by storing in the sent list, which reaches the last node tail. And tail of the destination bank gives the response message to the client as to the processing is done. The tail server of the origin bank also communicates to client with regard the transfer operation initiated. A contradiction for our assumption.

Suppose in case we have a head and a tail only, and the tail get the transfer request, it sends it to the head for processing. (It is aware of the head of the chain). Then the head processes the request and sends the result to the client. And since the tail is present, it sends a response to the client by the reliable TCP/IP connection (assuming that there are no loss of messages). Even if the message is lost, the request ID is already processed and another updated, so that the server knows if the same operation is already performed prior.

Suppose we have just a node in the server (this drops down to an un-replicated server case), then it acts both as the head and the tail in the chain. It receives the transfer request, processes it since it is the head. It replies back to the client about the operation because there is a reliable UDP communication between the client and the server to send the data. Therefore, at any point of time, the response after the transfer operation is conveyed to client without loss of transition. Thus our assumption that the client does not receive a response proves to be wrong. Hence, by proof by contradiction, the design successfully handles the transfer requests of the client effectively.