Sanket Dige (110815356)
Rashmi Makheja (110920870)

CSE 535: ASYNCHRONOUS SYSTEMS
Byzantine Chain Replication
Phase 1: Pseudo-code for HMAC Shuttle

Assumptions:

1) We assume that, at the start of the system, there is a configuration already in place. Every process will be given a set of configs directly like public-private keys of replicas, clients and olympus.
2) Each entity has the required public keys of other entities and its own public and private keys.
3) In case of client getting invalid result proof, we wait for correct responses by dropping the incorrect one. After receiving t + 1 incorrect responses, we consider it to be proof of misbehavior and send the reconfig request.
4) In case of replica getting invalid result proof or order proof, we consider it to be proof of misbehavior and send the reconfig request.

Message Types

Below are the different messages that we will be using throughout the pseudo code. Each message is associated with it's getter function which will generate the message and return a new Message object. The getters bundle all the message data in a map(key-value pair) so that it can be easily retrieved by the receiver using the keys.

i) <ORDER_MESSAGE>

// Message sent from client to the head to request result of an operation

```
{
    type : "order",
    sender_id : id of the client
    req_id : id assigned by client to uniquely identify new request from retransmission,
    operation : operation requested by the client
}
```

getOrderMessage(oper) : //this method returns a **<ORDER_MESSAGE>**

```
    //a variable incremented every time this function is called.
    // It maintains state throughout the execution
    req_id++
    return Map(
        "type" : "order",
        "sender_id" : this.ID // id of client object calling the method
        "req_id" : req_id,
        "operation" : oper
    )
```

---

ii) <ORDER_SHUTTLE_MESSAGE>

// Message sent from replica to another replica to forward the order shuttle

```
{
        type : “order shuttle”
        slot_no : slot-no
        operation : operation
        req_id : request id of the operation
        order_proof : order proof of the replica
        result : result of the operation obtained by this replica
        result_proof : {} result_proof of the operation
}
```

getOrderShuttleMessage(slot_no, oper, req_id, order_proof, res, res_proof)

// this method returns the **<ORDER_SHUTTLE_MESSAGE>**

```
        return Map(
                “type” : “order shuttle”
                “slot_no” : slot-no
                “operation” : oper
                “req_id” : req_id
                “order_proof” : order_ proof
                “result” : res
                “result_proof” : res_proof
        )
```

---

iii) <RETRANSMISSION_MESSAGE>

// Message sent from client to replicas to request result of an operation that it had sent earlier

// OR from replica to head to request to start a operation if replica doesn’t find the req_id in their cache

```
{
        type : “retransmit”
        sender_type : “client”/”replica” // used to identify who sent the request
        // denotes a client id if Sender_type == client else represents a replica id
        sender_id : id of the client/replica
        req_id : request if of the old message
        operation : operation to be executed
}
```

getRetransmissionMessage(sender_type, req_id, oper)

//this method returns the **<RETRANSMISSION_MESSAGE>**

```
        return Map(
                “type” : “reconfiguration”,
                “sender_type” : sender_type,
                “sender_id” : this.id // calling object’s id
                “req_id” : req_id,
                “operation” : oper
```

)

iv) <RESULT_MESSAGE>
//Message sent from replica to replica in a result shuttle OR
// from replica to client in case of retransmission OR
// from tail to replica to pass the result and result_proof of the operation

```
{
    type : "result",
    sender_id : //id of the replica sending the message.
    req_id : //request id of the operation,
    operation : //operation sent by client
    result : //result of the operation,
    result_proof : {} //Every replica adds the hash of its result to the result_proof
}
```

getResultMessage(req_id, operation, result, result_proof)
//this method returns a **<RESULT_MESSAGE>**

```
return map(
    "type" : "result",
    "sender_id" : this.id //calling replica's id
    "req_id" : req_id,
    "operation" : operation,
    "result" : result,
    "result_proof" : result_proof
)
```

---

v) <ERROR_MESSAGE>
// Message sent by replica to client in case of timeout

```
{
    type : "error"
    sender_id : id of the replica
}
```

getErrorMessage()
//this method returns the **<ERROR_MESSAGE>**

```
return Map(
    "type" : "error"
    "sender_id" : this.id // calling object's id
)
```

---

vi) <CONFIGURATION_REQUEST_MESSAGE>

// Message sent by client/replica to olympus to request for configuration

```
{
        type: "configuration req"
        ID : // id of the client/replica
        sender_type : client/replica
}
```

getConfigurationRequestMessage(sender_type)

// this method returns **<CONFIGURATION_REQUEST_MESSAGE>**

```
        return Map(
                "type" : "configuration req"
                "ID" : this.ID // calling object's ID
                "sender_type" : sender_type
        )
```

----

vii) <RECONFIGURATION_MESSAGE>

//Message sent from client/replica to olympus to request new configuration

// as the sender suspects faulty replica or error in the system

```
{
        type : "reconfiguration"
        sender_type : "client"/"replica" // used to identify who sent the request
        // denotes a client id if Sender_type == client else represents a replica id
        sender_id : id of the client/replica
        proof_of_misbehavior : //(result, result_proof) tuple to be sent by the client,
if sender is replica, then this will be null
}
```

getReconfigurationMessage(sender_type, result)

//this method returns the **<RECONFIGURATION_MESSAGE>**

```
        return Map(
                "type" : "reconfiguration",
                "sender_type" : sender_type,
                "sender_id" : this.id // calling object's id
                "proof_of_misbehavior" : result
        )
```

----

viii) <CONFIGURATION_MESSAGE>
// Message sent by olympus to client/replica to send current configuration

```
{
    type : “configuration”
    cur_config : current configuration of the system
}
```

getConfigurationMessage()
//this method returns **<CONFIGURATION_MESSAGE>**

```
    return Map(
        “type” : “configuration”,
        “cur_config” :   olympus.cur_config
    )
```

---

ix) <WEDGE_MESSAGE>
// Message sent by olympus to replicas to request for their history

```
{
    type : “wedge”
}
```

getWedgeMessage() //this method returns the **<WEDGE_MESSAGE>**

```
    return Map(
        “type” : “wedge”
    )
```

---

x) <WEDGED_MESSAGE>
// Message from replica to the olympus to send their history in response to **<WEDGED_MESSAGE>**

```
{
    type : “wedged”
    Sender_id : //id of the replica sending the message
    History : //history of the replica sending the message
    Checkpoint: //checkpoint of the replica sending the message
}
```

getWedgedMessage() //this method returns the wedgedMessage

```
    return Map(
        “type” : “wedged”,
        “sender_id” : this.id, //calling replica’s id
        “history”  : this.history, //calling replica’s history
        “checkpoint” : this.checkpoint, //calling replica’s checkpoint
    )
```

---

xi) <CHECKPOINT_MESSAGE>
// Message sent by replica's to other replicas when checkpointing proof is initiated by head

{
    Type: "checkpointing request",
    ID: "id of the replica",
    History: a list of order histories from each replica,
    Hash: a hash of the history set from each replica as a checkpoint proof
}

getCheckpointRequestMessage(history_set, hash)
// this method returns **<CHECKPOINT_REQUEST_MESSAGE>**

return Map(
    "History" : history_set
    "Hash": hash
)

----

xii) <CHECKPOINT_COMPLETE_MESSAGE>
// Message sent by replica's to other replicas when checkpointing is completed from tail and all replica's can delete their prefixes.

{
    Type: "checkpoint complete",
    ID: "id of the replica"
}

getCheckpointCompleteMessage(id)
// this method returns **<CHECKPOINT_COMPLETE_MESSAGE>**

return Map(
    Type,
    ID = id
)

----

xiii) <CATCHUP_MESSAGE>
// Message is send by olympus to quorum of replica

{
    Type: "catchup",
    Operation: "Difference of operations -> (Longest History - current replica history)"
}

getCatchupMessage(operations)

return Map(
"type": "catchup",
Operation: operations

}

----

xiv) <CAUGHTUP_MESSAGE>

// Message is send by replica to olympus

```
{
    Type: “caughtup”,
    Hash_state: “Hash of the state at replica”
}
getCaughtupMessage(Hash_state)
    return Map(
        “type” : “caughtup”,
        “Hash_state”: Hash_state
    )
```

---

xv) <GET_RUNNING_STATE_MESSAGE>

// Message is send by Olympus to replica in quorum

```
{
    Type: “get running state message”,
}
getGetRunningStateMessage()
    return Map(“Type”: “get running state message”)
```

---

xvi) <RUNNING_STATE_MESSAGE>

// Message is send from a replica to the olympus

```
{
    Type: “running state message”,
    RunningState: “Running state of the replica”

}
getRunningStateMessage(running_state)
    return Map(
        “Type”: “running state message”,
        “RunningState”: running_state
    )
```

Common Methods

---

i) H(message) : hash function used to check the integrity of the message
ii) E(key, msg) : encrypts message with the key
iii) De(key, encrypted_msg) : decrypts encrypted message using key

Pseudo-code for *Client*

<u>Objects:</u>
i) *client_config* : contains the client specific details
ID : unique id to identify the client
private_key : private key for client
public_key : public key for client

ii) *replicas_public_keys* : list of public keys of all replicas
iii) *olympus_public_key* : public key of olympus.
iv) *cur_config* : current config obtained from olympus that specifies the number of replicas, their order, head and tail of the current chain and quorum of replicas (correct representatives).

<u>Methods:</u>
i) executeOperation(*op*) : start a new transmission
Start timer
// gets current config from the olympus
send_message(olympus, **getConfigurationRequestMessage("client")**)

Await for **<CONFIGURATION_MESSAGE>** from the olympus.
order_msg = **getOrderMessage(*op*);**
encrypted_msg = **E(**replicas_public_keys(cur_config.head), order_msg**)**
Send encrypted_msg to head

Start a seperate thread to check for new config and return to main thread
Await for **<RESULT_MESSAGE>** response from the tail
decrypted_res = De(private_key, response)
if(**isValidResponse(decrypted_res)** == false)
//response is invalid and is received from tail
if(decrypted_res**.**sender_id == cur_config.tail)
//Send reconfig_req message to Olympus
reconfig_msg = **getReconfigurationMessage(**
"client", (result,result_proof))
encrypted_msg = E(olympus_public_key, reconfig_msg)
Send encrypted_msg to olympus

Await for **<CONFIGURATION_MESSAGE>** from the olympus.

```
                Restart the process
        //invalid response is not from tail
        else
                keep waiting as this was sent by a bogus replica

//start a retransmission
If timer expires before receiving response
        retrans_msg = getRetransmissionMessage(
                "client", order_msg.req_id, op)
        For each replica r
                encrypted_msg_r = E(replicas_public_keys[r], retrans_msg)
                send encrypted_msg_r to replica r

        found_response = false
        responses_count = 0
        invalid_responses = 0
        Await <RESULT_MESSAGE> i.e. response_r from replica r
        // if we havent yet found the valid response
        // and received t+1 invalid responses,
        // then it constitutes proof of misbehavior
        while(found_response == false &&
                responses_count != cur_config.replicas_count &&
                Invalid_responses <= cur_config.failures_handled + 1)
                )
                responses_count++
                // check if the received reply is a valid result
                if(isValidResponse(response_r, false))
                        found_response = true
                Else
                        invalid_responses++
        if(found_response)
                Discard all responses
        // received all the responses but haven't found the valid response
        // In this case, we take reconfiguration from olympus
        // and restart process.
        // There can be four follow up cases after restarting:
        // case 1: There is no configuration and client will end up in here repeatedly
        // In this case, we assume that at some point, configuration will come
        // and it will eventually get the result and end the process.
        // case 2: olympus will change the reconfiguration
        // and eventually client will get the result
        else
                reconfig_msg = getReconfigurationMessage(
                                "client", (result,result_proof))
                encrypted_msg = E(olympus_public_key, reconfig_msg)
                Send encrypted_msg message to Olympus
```

Await for **<CONFIGURATION_MESSAGE>** from the olympus.
Restart the process

ii) isValidResponse(response_r) :

```
// Every replica signs hash of it's result with it's private key
// and adds it to the result_proof
// We can decrypt it using the public key of the replica
// and thus validate it against the hash of received  result.
// If t + 1 values are correct, client accepts the result.
.
        correct_res_count = 0
        cur_hash = H(response.result)
        for each result_hash in response_r.result_proof
                if(cur_hash  == result_hash)
                        correct_res_count++
        if(correct_res_count >= cur_config.failures_handled +1)
                return true
        return false
```

Pseudo-code for *Replicas*

Objects:

i) *replica_config* : contains the replica specific details
    ID : unique id to uniquely identify replicas
    private_key : private key for replica
    public_key : public key for replica
ii) *replicas_public_keys* : list of public keys of all other replicas
iii) *olympus_public_key* : public key of olympus.
iv) *cur_config* : current config obtained from olympus that specifies the number of replicas, their order,
        head and tail of the current chain.
v) *clients_public_keys* : list of public keys of all clients
vi) *History* : list of all the order proofs of this replica and the replicas preceding it.
vii) *Result_cache* : list of all the (req_id, operation, result, result_proof) tuple
viii) cur_mode : PENDING, ACTIVE or IMMUTABLE // cur mode of the replica
ix) *persistent_order_no :* This variable will specify the index upto which order statements, the result shuttle has been received.
x) *checkpoint* :
    A list of checkpoint history.
xi) *prev_replia* : id of prev replica in the chain, null for head
xii) *next_replica* : id of next replica in the chain, null for tail

Methods:

i) getSlot()
```
    For i=presistent_order_no; i++
        If slot s at i is free
            return s;
```
ii) isHead()
```
    // if current replica is head
    if (ID == cur_config.head)
        return true
    return false
```
iii) isTail()
```
    // if current replica is tail
    if (ID == cur_config.tail)
        return true
    return false
```

iv) processRequest():
//this will be the main method that waits and receives all the messages of the olympus.
```
    Await for a message
    Decrypted_msg = De(msg)
```

```
    //Based on decrypted_msg .type, call appropriate function
    switch (decrypted_msg)
        case  <ORDER_MESSAGE> or <ORDER_SHUTTLE_MESSAGE> :
             executeOrder(decrypted_msg)
        case <RESULT_MESSAGE> : cacheResult(decrypted_msg)
        case <RETRANSMISSION_MESSAGE> :
            executeRetransmission(decrypted_msg)
        case <WEDGE_MESSAGE> : executeWedge(decrypted_msg)
        case <INIT_HIST_MESSAGE> : executeInithist(decrypted_msg)
        case <CHECKPOINT_MESSAGE>: executeCheckpointing(msg)
        case <CATCHUP_MESSAGE>: executeCatchup(msg)
        case<GET_RUNNING_STATE_MESSAGE>:
                    executeGetRunningState(msg)
```

v) executeOrder(msg)

```
// irrespective of the mode of the cache, if the result is found in cache, send the result from the cache
    If  result for operation found in result_cache
        return encrypted  cache entry
// if replica is in immutable state, don't send anything and drop the msg.
    If mode == IMMUTABLE
        return
// if head, get the slot, generate the order statement and the order proof
    if (isHead() == true)
        s = getSlot()
        o = msg.operation
        req_id = msg.req_id
        order_stmt = (s, o)

        // every replica signs the statement with its private key
        order_proof = signStatement(order_stmt, replica)
    // for any other replica, message has all the details
    else
        s = msg.slot_no
        o = msg.operation
        req_id = msg.req_id
        order_stmt = (s, o)
        order_proof = msg.order_proof

    // if valid order proof then make the result proof
    If isValidProof(msg.order, order_proof) == false
        //Send reconfig_req message to Olympus
        reconfig_msg = getReconfigurationMessage("replica", null)
        encrypted_msg = E(olympus_public_key, reconfig_msg)
        Send encrypted_msg to olympus
    else
        Perform operation o
```

```
        order_proof.add(signStatement(order_stmt, replica))
        result = result of operation o
        Result_stmt = signStatement(result)
        // if it's the head, then create a new result_proof
        if (isHead() == true)
                Result_proof = { Result_stmt }
        else
                result_proof.add(Result_stmt)
        // if tail, end shuttle and send the result to client and the prev_replica
        if(isTail() == true)
                res_msg = getResultMessage(
                                req_id, o , result, result_proof)

                encrypted_msg = E(clients_public_keys[msg.sender_id],
                                        res_msg)
                Send encrypted_msg to client
                encrypted_msg = E(replicas_public_keys[prev_replica],
                                        hashed_msg)
                Send encrypteded_msg to prev_replica
                result_cache.add((req_id, s, o, result, result_proof))
        // if any other replica, forward shuttle
        else
                shuttle_msg = getOrderShuttleMessage(
                        s, o, req_id, order_proof, result, result_proof)
                encrypted_msg =
                        E(replicas_public_keys[next_replica], shuttle_msg)
                Send encrypted_msg to next_replica
```

vi) executeRetransmission(msg):

```
        If (msg.req_id, msg.operation) in result_cache
                Get result from cache
                res_msg = getResultMessage(
                        req_id, operation, result, result_proof)
                encrypted_msg = E(clients_public_keys[msg.sender_id],
                                        res_msg )
                Send encrypted_msg to client
        Else If mode == immutable
                error_msg = getErrorMessage()
                encrypted_msg = E(clients_public_keys[msg.sender_id],
                                        error_msg )
                Send encrypted_msg to client
                return
        // Else if not head, forward request to the head and start timer
        Else if isHead == false
                Start timer
```

```
                encrypted_msg = E(replicas_public_key[head], msg)
                Send encrypted_msg to head
                If result_shuttle did not arrive before timer expires
                        err_msg = getErrorMessage()
                        encrypted_msg = E(clients_public_keys[msg.sender_id],
                                                err_msg)
                        Send encrypted_msg to client
                        return
        // do the following if replica is head
        else
                //if operation is found in history, wait for result_shuttle
                if  (msg.req_id, msg.operation) in history
                        Start_timer
                        If result_shuttle did not arrive before timer expires
                        err_msg = getErrorMessage()
                        encrypted_msg = E(clients_public_keys[msg.sender_id],
                                                err_msg)
                        Send encrypted_msg to client
                        Return
                // this operation seems unrecognized. So start entirely.
                else
                        Start timer
                        execute_operation(o)
                        If result_shuttle did not arrive before timer expires
                                err_msg = getErrorMessage()
                                encrypted_msg = E(
                                                clients_public_keys[msg.sender_id],
                                                err_msg)
                                Send encrypted_msg to client
                                return
```

vii) cacheResult(msg)

```
If isValidProof(msg.result, msg. result_proof) == false
        //Send reconfig_req message to Olympus
        reconfig_msg = getReconfigurationMessage("replica", null)
        encrypted_msg = E(olympus_public_key, reconfig_msg)
        Send encrypted_msg to olympus

        else
                Result_cache.add( [msg.req_id, msg.result, msg.result_proof] )
                Persistent_order_no++
                res_msg = getResultMessage(
                        req_id, operation, result, result_proof)
                encrypted_msg = E(clients_public_keys[msg.sender_id],
```

```
                                        res_msg)
        If timer is ON
                send_message(
                        client,
                        signStatement(encrypted_msg, replica)
                )
                Stop timer
        encrypted_msg =
                E(replicas_public_key[prev_replica], hashed_msg)
        send_message(
                prev_replica,
                signStatement(encrypted_msg, replica)
        )
```

viii) executeWedge()

```
becomeImmutable()
W = {this.History, this.checkpoint}
w_msg = getWedgeMessage(W)
encrypted_msg = E(olympus_public_key, w_msg)
Send encrypted_msg to olympus
```

ix) executeInithist(msg)

```
        For each <s,o> from msg.history
                If each<s,o> not in this.history
                        Perform operation o
        becomeActive()
```

x) signStatement(statement, replica)

```
Return E(replica.private_key, H(statement))
```

// This function will be running on a separate thread at head replica that will periodically start checkpointing

xi) startCheckpointing()

```
If replica == head
        History = []
        checkpoint_history = { }
        last_checkpoint = last checkpoint in History
        For each_order beyond last_checkpoint in History
                checkpoint_history.add(order)
        history[r] = checkpoint_history
        // Send message signStatement(statement, replica) to next_replica
        send_message(
                next_replica,
                getCheckpointRequestMessage(history, H(history))
        )
```

xii) executeCheckpointing(msg)

```
Decrypted_msg = De(msg, replica.private_key)
if(decrypted_msg == msg.hash)
    History = []
    checkpoint_history = { }
    last_checkpoint = last checkpoint in History
    For each_order beyond last_checkpoint in History
        msg.history[r].add(order)

    // if its tail, completed checkpoint proof is send back to all replicas
    // to remove the corresponding prefix of the history
    If replica == tail
        statement.type = complete_checkpointing
        statement.history = msg.history
        Send signStatement(statement, replica) to previous_replica
        send_message(
            previous_replica,
            getCheckPointCompleteMessage(replica.id)
        )
    Else
        send_message(
        next_replica,
        getCheckpointRequestMessage(msg.history, H(msg.history))
    )
```

xiii) executeCatchup(msg)

```
Decrypted_msg = De(msg, replica.private_key)
Execute operations : Decrypted_msg.operations
Hash_state - H(replica state)
send_message(olympus, getCaughtUpMessage(Hash_state))
```

xiv) executeGetRunningState(msg)

```
Decrypted_msg = De(msg, replica.private_key)
send_message(olympus, getRunningState(State S of replica))
```

xv) isValidProof(cur_stmt, proof)

```
// Every replica signs hash of it's order_stmt or result with it's private key
// and adds it to the order_proof and result_proof respectively
// We can decrypt it using the public key of the replica
// and thus validate it against the hash of received order statement or the result.
// If any replica has a different value, then it constitutes as proof of misbehavior
    For i = 0 to (cur_config.replicas_count - 1)
        stmt = decrypt(replicas_public_keys[i], order_proof[i])
        if(order_stmt != cur_stmt)
            return false
```

return true

Pseudo-code for *Olympus*

Objects:

i) olympus_config : contains the olympus specific details
    private_key : private key for olympus
    public_key : public key for olympus
ii) replicas_public_keys : list of public keys of all replicas
iii) replicas_private_keys : list of private keys of all replicas
iv) clients_public_keys : public keys of all client
vi) client_private_keys : private keys of all client
vii) caughtupMessageHash: save cryptographic hash of caughtup from quorum replica
iv) cur_config :
    config_id = sequence number of configuration
    failures_handled = t
    replicas_count = (2 * failures_handled +1)H
    replicas = {} //List of replica objects
    Head = //head of the chain
    tail = //tail of the chain

Methods:

i) processRequest():

//this will be the main method that waits and receives all the messages of the olympus.

```
decrypted_msg = De(private_key, msg) // Decrypt a received message
//Based on decrypted_msg .type, do appropriate processing
If validateMsg(decrypted_msg) == false
    Drop the message
else
    switch (decrypted_msg)
        case <CONFIGURATION_REQUEST_MESSAGE> :
            config_msg = getConfigurationMessage()
            Encrypted_msg = E(
                clients_public_keys[decrypted_msg.sender_id],
                config_msg)
            Send encrypted_msg to client

        case <RECONFIGURATION_MESSAGE> :
            processReconfiguration()
```

```
                    case <WEDGED_MESSAGE> : processWedged(decrypted_msg)
                     case<CAUGHTUP_MESSAGE>:
                            processCaughtUp(decrypted_msg)
                    case <RUNNING_STATE_MESSAGE>:
                        processRunningStateMessage(msg)
```

ii) processReconfiguration(msg):

```
    if(msg.sender_type == client)
        if(validateReconfigurationRequest(msg))
            For each replica r
                Signed_wedge =
                 E( replicas_public_keys[r], getWedgeMessage())
                sendMessage( r, signed_wedge)
```

iii) processWedged(msg)

```
    isValidWedged(decrypted_msg) == true
        verifyCheckpointProof(decrypted_msg.checkpoint)
        Save checkpoint
        W = []
        LH = {}
        new_quorum = cur_config.quorum
        if( decrypted_msg.replica is in cur_config.quorum )
            isConsistentHistory(decypted_msg.history)
            If LH is not longest compared to decypted_msg.history
                LH = decypted_msg.history
            Add decypted_msg.history in w
        Else
            // Need to choose new quorum
            new_quorum = new Quorum();
            if(w.size == cur_config.quorum.size)
            // construct history h by

            For eachReplica r in cur_config.quorum
                send_message(r, getCatchupMessage(LH - W[r]))
            // choosing longest order proof for each slot
                cur_config = new Config()
                cur_config.history = h
                cur_config.quorum = new_quorum

                //before the inithist we need to call get_running_state
                // for a replica in quorum
                send_message(any_replica_in_quorum,
            getGetRunningStateMessage())
```

iv) isConsistentHistory(history)

```
    For every pair of replicas r1 r2:
        For each slot s
```

```
                // same slots should have same operations
                if( !History[r1][s] = History[r2][s])
                        Return false
        Return true;
```

v) processCaughtUpMessage(msg)

```
        caughtupMessageHash == “”
                caughtupMessageHash = msg.hash_state
        else If msg.hash_state != caughtupMessageHash
                Cur_config.quorum = new_quorum
```

vi) processRunningStateMessage(msg)

```
        Decrypted_msg = De(msg, privateKeyReplica)
        If H(Decrypted_msg.RunningState) == caughtupMessageHash
                H = Add running State S in Inithist h
                For each replica r
                        sendMessage(r, H)
        Else
                send_message(other_replica_in_quorum, getGetRunningStateMessage())
```

vii) validateReconfigurationRequest(response)

```
// Every replica signs hash of it’s result with it’s private key
// and adds it to the result_proof
// We can decrypt it using the public key of the replica
// and thus validate it against the hash of received  result.
// If t + 1 values are correct, client accepts the result.
.
        incorrect_res_count = 0
        cur_hash = H(response.result)
        for each result_hash in response_r.result_proof
                if(cur_hash  != result_hash)
                        incorrect_res_count++
        if(incorrect_res_count >= cur_config.failures_handled +1)
                return true
        return false
```