BYZANTINE CHAIN REPLICATION

PSEUDO CODE

SUBMITTED BY

NAME	ID
ANKUR MAHESHWARI	110008503
VIPUL GANDHI	111483224

Olympus :

```
/* This variable will always store current configuration (I.e the Active one). Current
config gets changed after reconfiguration. */

CurrentConfig C


/*This function creates public and private signing cryptographic keys for Client and
Replicas. After creation Olympus sends them to respective Replicas and Clients */

def createKeysforReplicasandClients

       // Olympus gets the config size i.e 2t+1 size of the configuration
       int configSize = C.getConfigSize();



       /*Cyrptographic private and public N (configSize)signing keys, verify keys are
       generated*/
       for (i=1 to  configSize)
              privatesignkey(i) = signingkey.generate()
              publicverifykey(i) = privatesignkey(i).verify_key



       //Private and Public signing keys are generated for client
       privatesignclient = signingkey.generate()
       publicverifykeyclient = privatesignclient.verify_key



       //All replicas are get from Configuration
       Array<Replica> aR  = C.getAllReplicas()




       /*Public and private keys are sent to respective Replicas so that they can sign
       and verify when order statements are transmitted*/
       for (i=1 to  configSize)
              send(“privatesignkey”,privatesignkey(i) to aR(i))

              for(j=1 to  configSize)
                     if(i!=j)
                            send(“publicverifykey”,publicverifykey(i) to aR(j))



       /*Public keys of all replicas are sent to Client so that Client can verify
       statements issued by replica to client*/
       for (i=1 to  configSize)
              publicverifykey =  publicverifykey + publicverifykey[i] + “,”

       send(“publicverifykey”,publicverifykey to client)
```

```
/*Client public key is sent to all Replicas so that Replica can verify messages
sent by Client*/
for(i=1 to  configSize)
      send(“publicverifykeyclient”, publicverifykeyclient to aR(i))



//Private key for Client is sent to client so that it can sign the statements
send(“privatesignclient”,privatesignclient to client)
```

```
/*Function checking if all replicas in chosen Quorum Q are consistent or not. Returns
false as soon as one Replica in Quorum is faulty.*/
def consistentReplicaforQuorum(WM(q),Q)
       // Qsize is the size of Quorum
       int Qsize = Q.size()
        /* slotAndOperationMap<slot,operation> Map for each replica in Quorum is
       derived from wedgeStatement’s history in current Quorum */
      for all Wi in WM(q)
                slotAndOperationMap<s,o>[i] = getSlotsAndOperations(Wi.hist)


       bool iscorrectQuorum = true

       for(i=1 to  Qsize)
             for(j =1 to Qsize)
                   if(i!=j)
                         for(key s in slotAndOperationMap<s,o>[i])
                         /*slotAndOperationMap[s][i] is o(operation)(Value
                         corresponding to the key s}*/
                            if(slotAndOperationMap[s][i]!=slotAndOperationMap[s][j])
                                     /*For the pair of replicas having slot s ,
                                     they have different values of operations(e.g
                                     o and  o’)*/
                                     iscorrectQuorum = false




       if(iscorrectQuorum == true)
             return true
       else
             return false
```

```
/* Flag maintaining whether a valid Quorum is found yet or not.Used in many
functions*/
flag keepSearchingForQuorum = true


/*The below function sets keepSearchingForQuorum to false if Quorum is found to have
consistent replicas*/.
def checkAcceptableQuorum(W(q),Q)
        if(!consistentReplicaforQuorum(W(q),Q))
                keepSearchingForQuorum=true
        else
                keepSearchingForQuorum=false

//This function returns a combination of t+1 replicas out of the total replicas
def chooseQuorum(Q,t+1)
        /*It selects random t+1 replicas to be part of Quorum Q*/
        return choose(random(Q,t+1))




/*Olympus receives sends ReConfigRequest from Client, Replica and signed wedgeRequests
to all replicas in the current configuration*/
receive ("ReConfigRequest" from Client,Replica)
        //lR is collection of all replicas in current configuration C
        List<Replica> lR = C.getAllReplicas()

        //Send wedgeRequest to all replicas in lR
        //r represnts the replica to which current wedgeRequest is being sent
        for  r in lR
                signedWedgeRequest = privatesignolympus.sign(r)
                send ("wedgeRequest", signedWedgeRequest, to r)







//Rc   all replicas which would send wedgeStatements
Rc = {}
//Q is the Quorum for set of chosen non-faulty replicas for current configuration C
Q = {}

/*Backbone code for Olympus. Builds Rc. Forms Qc. Validates Q. Forms
LongestHistory(LH) and sends catchup message to Replicas in Q.*/

receive ("wedgeStatement", WM from Replica)

        /*WM(r) is the list of WedgeStatements of all replicas which have replied with
        WedgeStatements in response to Olympus wedge statements*/

        WM(r) = WM(r) U {WM}


        /*A loop for choosing Quorum and keep searching for one if we don’t get
        Validated Quorum*/
        do{
                //Q is the Quorum
                Q = chooseQuorum(Rc)
                //WM(q) are  WedgeStatements corresponding to chosen Quorum
                checkAcceptableQuorum(WM(q),Q)
        }
        while(keepSearchingForQuorum)



        //LH is the LongestHistory for all replicas in Q
        LH = {}
        for all W in WM(q)
                //updated LH to be max history of any replica in choosen Quorum Q
                LH = max(LH, W.hist)



        //For every replica in Quorum,send its deficit history suffic in catchup msg
        for all Wr in WM(q)
                if(W.hist < LH)
```

```
                        send ("catchup", r, {LH-Wr.hist} to r)
```

```
/*List stores received ch of all replicas {ch = cryptographic hash of runningState
sent by replica}*/
chList = {}

/* Only for a Consist/Valid quorum, Cryptographic Hash of the state sent by
replicas(with same state) */
finalCh


/*Olympus checks if it receives the same cryptographic hash of runningState in the
caught_up messages from all replicas in Q. Else, again a new quorum is formed.*/

receive ("caught_up", ch from Replica)
        //add received ch to the  chList
        chList.add(ch)
        /*Cryptographic hash of runningState for all replicas should be same. If any
        ch does not match, one of the replica in Q is faulty*/
        for i= 0 to chList.size()-1
                if(ch[i] != ch[i+1])
                        QuorumFound=false /*ch mismatches. Faulty replica found.
                                        Search for a new Quorum*/


        QuorumFound=true

        if(QuorumFound==true)
                /*update global variable finalCh to ch(same for all replica in
                Quorum Q)*/
                finalCh = ch
                /*Now ask any replica for its runningState just to make sure that the
               runningState has not changed meanwhile the Quorum choosing process.*/
                send ("get_running_state",r to Replica)
        if(QuorumFound==false)
                chooseQuorum(Q) //Choose another set of t+1 replica
```

```
receive ("sendRunningState" , S from Replica)
        from Wi.History
                /*Check Cryptographic hash of the received state S from any replica
                should match the earlier saved runningState finalCh(Last step before
                making initHist() call at Olympus end)*/
                if(CryptoHash(S)==finalCh)
                    //History of next config C' is empty at this moment
                    initHist(S)
                else
                    //Again ask any other replica for its runningState
                    send ("get_running_state",r to Replica)
```

```
/*This function instantiates fresh (2t+1)replicas and seeds their runningState with
state S*/
def inithist(S)
        //Assign the received runningState S to the new configuration
        C.setRunningState(S)
        // assign a total of 2t+1 fresh replicas to new configuration
        C.assignReplicas(2t+1)
```

Client :

executeOperation()

```
       // Getting the next operation to be performed on Configuration from Application
       Operation Op = get next request from application

       /*For liveness, the client checks periodically with Olympus to see if there is
       a new configuration, and if so retransmits its operation */
       for(after every time t' interval)

              //Send and receive the Current Configuration from Olympus
              send("getCurrentConfig" to Olympus)
              receive("getCurrentConfig" , C from Olympus)

              //If C has Changed, then re-transmit request to Head
              if(C has changed)
              send("request", signedorder,order,1 to headR)
```

```
    /*Since the configuration is chain of 2t+1 replicas. It will have all details
    about head, no of replicas etc. so we get the head of the configuration C  */
    Replica headR = C.getHead();

    //This flag is set to 1 when request is re-transmitted
    retransmitFg = 0

    /*The statement that has to be sent has following information.op-operation,
    Id is unique request Id sent,C is Configuration and headR is head replica*/
    orderMsg = op,headR,C,Id

    // Statement is signed before sending it to head from client
    signedorder =  privatesignclient.sign(op)

    //Order is sent from client to head
    send(“request”, signedorder,orderMsg,retransmitFg to headR)

    //CLIENT RECEIVER 1 (FROM REPLICA)
    //Client waits for result and resultproof of the operation performed
    receive(“request”,signedResultProof,ResultProof, result from Head Replica )


    /*If resultproof gets returned successfully before timeout then validation on
    resultproof are performed by Client*/

    await  ResultProof :
             publicverifykey(r).verify(signedResultProof)
            //Validate ResultProof
            bool resvalidate = true
            for(rp in ResultProof)
                    //Comparing  crphash(result) with ResultProof’s Result Statement
                    if(crphash(result) != rp.(crphash(r)))
                    resvalidate = false
                    break;

            if(resvalidate== false)
            send(“reconfigrequest” to Olympus)


    /*The timer is initiated for the request.If the timeout happens I.e the result
    is not returned before  timeValue, then Client sends request to all Replicas*/
    or
    timeout timeValue :
            Array<Replica> lR = C.getAllReplicas()

            retransmitFg = 1

            for(ri in lR)
                    msg = op,ri,C,Id
                    signedmsg = privatesignclient.sign(stmt)
                    send(“request”, signedorder,orderMsg,retransmitFg to ri)

                    //Client error from the Replica
                    receive(“Error”,ErrorMsg from Replica )

                    //Client Try Again.Order is sent from client to head
                    send(“request”, signedorder,orderMsg,retransmitFg to headR)


            //Client waits for result and resultproof of the operation performed
            receive(“request”,signedResultProof,ResultProof, result from Replica )



//Receiving public and private keys from Olympus
receive(“privatesignclient”, privatesignclient from Olympus)
receive(“publicverifykey”, publicverifykey from  Olympus)
```

Replica :

```
History hist               //It denotes the history I.e collection of order proofs
//Order Proofs of the previous replicas
OrderProof orderProof [s,o,r,C,list<OrderStmt>signedorders]
OrderStmt<s,o>(r)    //Order Statement of current replica
Slot s              // s for which operation should be unique
ResultProof resultProof    //ResultProof comes back in result shuttle

//For Caching ResultProof and Result when result shuttle comes back
Map[Id,CachedProof[ResultProof,Result]] IdMap

//When max size if achieved.Over that,it will delete Least First Entry inserted
maxSizeMap

//Checkpoint would be applied after cN interval of slots
CheckPointInterval cN

//CheckPointProof for storing checkpint and crpytographic hash of state
CheckPointProof[checkpt,crpythash(state)] chkptProof

//Private and Public keys for signing and Verifying
privatesignkey(r)
publicverifykey(1 to C.getRepliSize())

/*order and signed are are sent from client to replica, replica to replica
order*/
signedorder


/*To prevent holes, a replica should sign an order statement for a slot only if it has
signed order statements for all lower-numbered slots . This function takes care that
slot allocated is not lower or equal than previous slots*/
```

def validateSlot(int s)

```
      for(slot s(i) in hist.getAllSlots)
            if(s(i) >= s)
            return false

      return true
```

```
/*An active replica ρ can suspend updating its history by becoming immutable at any
time.The replica signs a wedged statement to notify Ω that it is immutable and what
its history is*/
```

def becomeImmutable(Replica r)

```
      if(r.mode == ACTIVE)
            r.mode = IMMUTABLE
            orderProof.add(wedged,r.hist)
```

```
/*Any active replica r  in configuration C can say <order, s, o>(r)  if each preceding
```

```
replica in C, if any, has done likewise and there is no conflicting operation for s in
its history. r also adds a new order proof to its history*/
//r:current replica,C: Current Configuration,s: slot number , op : operation
def  orderCommand(r,C,s,op)

    bool precond1 = true, precond2 = true
    if(r belongs to C && r.mode == ACTIVE)

        for(orderstmt(r') in   orderProof.signedorders)
            /*To check that the order proofs of preceding replicas are signed
            correctly. r' is used for preceding replicas to r */
             publicverifykey(r').verify(orderstmt(r'))

             /* <s,op>(r')is taken from  orderproof(r') i.e for the previous
              replicas */
             if(<s,op>(r) not equals  <s,op>(r'))
                    precond1 = false

        /*If the  replicas Order Proofs with same slot and different
        operation exits in history, then also transaction cannot be processed*/
        if( <s,op',r',C,<s,o>(r') belongs to r.hist)
            precond2 = false


    /*If the both the precondition are satisfied then add Order to Order Proof
    and add Order Proof to the History*/

    if(precond1 && precond2)
        order =  <s,op>
        OrderStmt(r) = privatesignkey(r).sign(order)
        orderProof.signedorders.add(OrderStmt(r))
        r.hist.add(orderproof)//equivalent to r.hist.add(s,op,r,C,sign(order)
    /*else send reconfiguration request to Olympus.Misbehavior has been detected*/
    else
        send("ReConfigRequest" to Olympus)
```

As stated in project.txt

the paper does not discuss how replicas detect provable misbehavior; you should think about this and explicitly discuss it in your submission.

We have not included in the Pseudo code. So, we are explicitly mentioning it as a separate section. We can take below situations into consideration for provable misbehaviors. As per Solution found in Chi Ho's thesis : Reducing costs of Byzantine fault tolerant distributed applications.pdf
A member runs the failure detection protocol when it is active when following two events happen :
1. Corrupted messages: A message that has been authenticated to have been sent from a sender but its contents is not what it should be. For example, a message carrying a malformed order proof and verified to have been sent by the leader indicates that the leader has failed. Note that a message that cannot be authenticated does not raise a detection or suspicion, as it may have been sent from an attacker in the network.
2. Lost messages: A member expects some particular messages in each step of the protocol. When the member times out on waiting for a message, it raises a suspicion. For example, during the processing of a message m, • a member can time out on waiting for an order proof hRp, {hm, ci i | i ∈ Rp}i after it has signed hm, ci; • a leader can time out while collecting signatures for an order proof hRp, {hm, ci i | i ∈ Rp}i; • and so on... When a member i of Gp raises an FDS event, it changes statep i from active to passive. The replicated server progresses no further. And eventually other members will change their state from active to passive as well.

```
/* It is the general function used to check the slot is for checkpointing or not and
then forward the shuttle to next replica. Also takes care of  sending back result to
Client and return shuttle,checkpoint shuttle  to previous replicas from Tail.
It will handle head to next Replica, Replica to Replica shuttle.These functions use
object properties and data member to compute */

def transmitShuttle()
    //This checks whether this s should be indentified for checkpointing
    if(s % cN == 0)
        /*adding checkpoint and cryptographic hash of the running state to the
        checkpoint   proof */
        chkptProof.add(checkpt,crpythash(state))



    //applies o to its running state and obtains a result
    Result result = applyOperation(state,op)

    /*adds <order,s,op> to the order proof (the replica undergoes an orderCommand
    transition)*/
    orderCommand(r,C,s,op)

    /*adds <result, op, hash(r)>ρ to the result proof, where hash is a
    cryptographic hash function*/
    signedresultproof = privatesignkey(r).sign(result,op,hash(result))
    resultProof.add(signedresultproof)

    //Gets the next Replica from Configuration
    Replica nr = C.nextReplica()

    //If the next Replica exits and therefore current Replica is not Tail
    if(nr != null)
        ordermsg = op,s,r,C,Id
        signedorderstmt = privatesignkey(r).sign(<s,op>)
        send(“request”,signedorderstmt, ordermsg to nr)

    //Tail is reached (Current Replica is Tail)
    else

        prevReplica = C.prevReplica(r)

        /*Tail forwards the result proof to the client along with the result
        itself.*/
        send(“request”, signedresultproof,result to Client)



        /*The tail also returns the shuttle with the completed proofs to the head
        along the chain in the reverse order*/
        if( prevReplica != null)
            send(“returnrequest”, signedresultproof,result,prevReplica,C,Id to
            prevReplica)


            /*The tail returns the completed checkpoint proof along the chain
            so each replica can remove the corresponding prefix from its
            history*/

            if(s % cN == 0)
                send(“returnchkrequest”, chkptProof,prevReplica to
                prevReplica)
```

```
/* This function handles all 3 cases of re-transmission to Head  I.e
1.it has cached the result shuttle corresponding to the operation;
2. it has ordered the operation but is still waiting for the result shuttle to come back;
3. it does not recognize the operation.*/

def handleReTransmissionRqtToHead()

    //1.Checking if the request id is already IdMap , then sending result  to Client
    if(Id exits in IdMap)
        resultproof =  IdMap[Id].CachedProof.ResultProof
        result =  IdMap[Id]. CachedProof.Result
        signresproof =  privatesignkey(r).(resultproof)
        send(“request”,  signresproof,result to Client)

    /*2.At Head,Request has been ordered and waiting for the
    result to be cached */
    if(orderProof has order<s,o> && IdMap[Id].CachedProof==null)

        //It waits for result to come back before timeout
        await IdMap[Id].CacheProof:
            cancel timeout
            resultproof = IDMap[Id].ResultProof
            result = IDMap[Id].Result
            signresproof =  privatesignkey(r).(resultproof)
            send(“request”,  signresproof,result to Client)

        timeout timeValue: //Send ReConfiguration to Olympus
            send(“ReConfigRequest” to Olympus)

    /*3.Re-transmitted Operation not recognised by Head*/
    else if(Id not found in IdMap like Operation Unrecognised)

        /*Head  increases the slot
        transmits the shuttle forward */
        s = s + 1
        transmitShuttle()
        //It starts the timeout and waits for result to come back before timeout
        await IdMap[Id].CacheProof:
            cancel timeout
            resultproof = IDMap[Id].ResultProof
            result = IDMap[Id].Result
            signresproof =  privatesignkey(r).(resultproof)
            send(“request”,  signresproof,result to Client)
        timeout timeValue: //Send ReConfiguration to Olympus
            send(“ReConfigRequest” to Olympus)
```

```
//REPLICA RECEIVER 1 (FROM CLIENT)

//Receiving the signed  operation from client to Replica and Handling it

receive(“request”,signedOrderStmt,orderMsg,retransmitFg from Client)


    /*Verification of signedOrderProof is done in orderCommand also*/

    //Verify the client has signed correctly
    publicverifykeyclient.verify(signedOrderStmt)

    /* Split the statement into op – Operation, previous replica , C –
    Configuration, Id : Unique Id of request */
    [op,prevr,C,Id] = orderMsg.split(,)

    r = C.nextReplica(prevr)

    //Checking if the request id is already IdMap , then sending result  to Client
    if(Id exits in IdMap)
        resultproof =  IdMap[Id].CachedProof.ResultProof
        signresproof =  privatesignkey(r).(resultproof)
        result =  IdMap[Id]. CachedProof.ResultProof
        send(“request”,  signresstmt,result to Client)




    else
        headR = C.getHead()
        //If the current Replica is Head (which has received rqt from Client)
        if(r ==  headR)
            //Head is called first time from Client
            if( retransmitFg == 0)

                /*Head is called first time ,it increases the slot.*/
                s = s + 1


                /*To prevent holes, a replica should sign an order
                statement for a slot only if it has signed order statements
                for all lower-numbered slots */
                while(validateSlot(s) == false)
                s = s + 1

                //Head transmits the shuttle forward
                transmitShuttle()


            //Head is called second time from Client after timeout at Client
            else if( retransmitFg == 1)
                 //Function definition is specified above
                 handleReTransmissionRqtToHead()



        //If  Replica which has received request from Client
        else
            /*If the replica is immutable, it responds to the client with an
            error statement*/
            if(r.mode == IMMUTABLE)
                send(“ErrorRequest”, “Error its Immutable” to Client)

            /*Replica sends request to Head and starts a timer.Note that Cache
            check I.e IdMap has already placed at the start of handler*/

            else
                //Replica sends request to Head and starts Timer.
                ordermsg = op,s,r,C,Id


                signedorder = privatesignkey(r).sign(orderMsg)
                send(“request”, signedorder, orderMsg to headR)

                receive(“request”,signedResultProof,ResultProof,result from
                headR)
                pubicverifykey(headR).verify(signedResultProof)
                await  ResultProof:
                    /*If request is received , then cancels timer and
                    send  ResultProof and result back to Client*/
                    cancel timeout
                    signedresultproof = privatesignkey.sign(ResultProof)
                    send(“request”,signedresultproof,result to Client)

                timeOut timevalue :
                    send(ReConfigRequest” to Olympus)
```

//REPLICA RECEIVER 2 (FROM REPLICA I.E REPLICA TO REPLICA)FORWARD SHUTTLE

```
//Receiving the signed  operation from Replica to Replica and handles FORWARD shuttle

receive(“request”, signedOrderStmt,orderMsg  from Replica)

    /* Split the statement into op – Operation, previous replica , C –
    Configuration, Id : Unique Id of request */
    [op,prevr,C,Id] = orderMsg.split(,)

    //Verify the previous replica was signed correctly
    publicverifykey(prevr).verify(signedOrderStmt)
```

```
        r = C.nextReplica(prevr)
        headR = C.getHead()


        /*If receiver Replica is not head , this will be just Replica to Replica
        shuttle*/
        if(r != headR)
                transmitShuttle()


        else
              //If the receiver Replica is head, handle all 3 cases as specified in fxn
               handleReTransmissionRqtToHead()
```

//REPLICA RECEIVER 3 (FROM REPLICA I.E REPLICA TO REPLICA)RETURN SHUTTLE

```
//Receiving the signed  operation from Replica to Replica and handles RETURN shuttle

receive("returnrequest",signedResultProof,ResultProof,result,r,C,Id  from Replica)


        /*Note that nR is Next Replica and pR is previous Replica*/
        Replica pR = C.prevReplica(r)
        Replica nR = C.nextReplica(r)

        //Verification of  signedResultProof
        publicverifykey(nR).verify(signedResultProof)


        signedResultProof  = privatesignkey(r).sign(ResultProof)

        //Result is cached when result shuttle of complete replicas returns
        IdMap.insert(Id,CachedProof[ResultProof,Result])

        /*If max size is achieved, then delete First Entry(FIFO fashion). IdMap is used
        to store request id s */
        if(IdMap.size() == maxSizeMap)
        IdMap.deleteFirstEntry()


        send("returnrequest", signedResultProof, ResultProof, result,pR,C,Id to pR)
```

//REPLICA RECEIVER 4 (FROM REPLICA I.E REPLICA TO REPLICA)CHECKPOINTPROOF SHUTTLE

```
//Receiving the signed  operation from Replica to Replica and handles forward shuttle
```

```
receive(“returnchkrequest”,chkptProof,C,r  from Replica)


       //Update with complete  checkpoint proof which returns from Tail to Head
        chkProof(r).update(chkptProof)


       // each replica  removes the corresponding prefix from its history
       hist(r).empty()

        /*chkptProof(r) is signed to be sent back to prev Replica(note that for back
       shuttle it’s actually next) */
       Replica pR = C.prevReplica()


       send(“returnchkrequest”,  chkptProof,C,pR to pR)
```

//REPLICA RECEIVER 5(REPLICA RECEIVES KEY FROM OLYMPUS)

```
receive(“signverifykeys”,privatesignkey,publicverifykeys[],C from Olympus)

       //Receiving the private sign key in data member of Replica
       privatesignkey(r) =  privatesignkey

       for(I in C.getAllReplicas where I != r)
       publicsignkeys(I) =  publicverifykeys[]
```

//REPLICA RECEIVER 6(REPLICA RECEIVES signedWedgeRequest FROM OLYMPUS)

```
receive ("wedgeRequest",signedWedgeRequest, r from Olympus)

        /*verify signedWedgeRequest. If verification turns out false, exception is
         raised and below code is not executed.*/
        publicverifyolympus(r).verify(signedWedgeRequest)

        //Make the current replica immutable. No more operations.
        if(r.state = "ACTIVE")
                r.state = "IMMUTABLE"


        //Add current history and most recent checkpoint proof to the wedgeStatement
        w = {}
        w.hist = r.hist
        w.chkptProof = r.chkptProof


        /*Our wedgeStatement comprises of only history and checkpoint proof. */
        send ("wedgeStatement", w to Olympus)
```

//REPLICA RECEIVER 7(REPLICA RECEIVES PendingHistory FROM OLYMPUS)

```
/*Replicas of the current Quorum need to catchup to LongestHistory. Replica will
execute pending operations. Get current runningState and send cryptographic hash of
runningState state back to Olympus.*/
receive ("catchup", r , PendingHistory from Olympus)
```

```
        /*get the list of all the pending operations to be executed to calculate the
        running state.*/
        operationList = getOperations(PendingHistory)
        //execute all operations in the  operationList
        for all operation in operationList
                applyOperation(operation)
        //get current runningState
        currentState = getRunningState(r)
        //Send cryptpgraphic hash of the running state back to  Olympus
        send ("caught_up", crypthash(currentState) to Olympus)
```

//REPLICA RECEIVER 8(REPLICA RECEIVES getRunningState FROM OLYMPUS)

```
receive ("get_running_state", r from Olympus)
        send <"sendRunningState" , r.State> to Olympus
```