Data Structure Format:

1. **Configuration:**
   # Contains seq num of the configuration. Because olympus keeps multiple configuration.
   # Ordered list of replica identifiers.
   # List of quorum.
   seqNumber
   Replica replicaList = []   #2t+1 replicas
   Quorom quoromList = []

2. Quorom:
   # Each Quorom has unique id and list of atleast t+1 replica.
   quoromID
   Replica replicaList = []   #atleast t+1 replicas

3. orderProof:
   <s, o, rho, C, {<order, o, s> …….. }>
   s - slot number
   o - operation. It is a tuple of <operationType, operationIdentifier>
   rho - replica
   C - configuration
   orderStatement orderStatements[]   # Cumulative order statements till the present rhoID (replicaID)

4. orderStatement:
   It is a tuple of <order,o,s>
   s - slot number
   o - operation it is a tuple of <operationType, operationIdentifier>

5. resultProof:
   <s, o, rho, C, {<result, o, S(r)> …….. }>
   s - slot number
   o - operation. It is a tuple of <operationType, operationIdentifier>
   rho -replica
   C - configuration
   resultStatement resultStarements[]   # Cumulative result statements till the present rhoID

6. resultStatement:

It is a tuple of <result,o,S(r)>
s - slot number
o - operation it is a tuple of <operationType, operationIdentifier>
S(r) - S is a cryptographic hash function. It computes hash of result.

7. checkpointProof :

# It is a list of checkpointStatements
# Every replica will add checkpoint statement to checkpointProof
checkpointStatement checkpointList[]

8. checkpointStatement

# it is a tuple of <checkpoint, S(state)>
S(state) - cryptographic hash of running state.

9. wedgedStatement:

It is history of replica which each replica will send to olympus.

10. shuttle:

# Contains
# 1. orderProof of operation o
# 2. resultProof of operation o
# Every replica will append order proof and result proof and checkpoint proof if checkpoint is initiated by head to shuttle and will send shuttle to next replica.

11. operation(o):

# it is tuple operation type and unique identifier **operationIdentifier received from client.**
operationType
operationIdentifier

12. inithistStatement:

It is tuple of <inithist, C, history>

Client -

class client(process):

```
        # Client receives operation request from application.
        # Client creates a unique identifier for each operation.
        # Client fetches active configuration from olympus.
        # Client keeps queue to store all incoming request.
        # ClientID is used to differentiate among different clients. Every client will have it's own
client id. This can be generated by calling createUniqueID function which consists of list of
unused Ids. This function will get called only once, when client setup is done for first time.
        clientID = createUniqueID()

        # OperationID is attached with clientID to create a unique identifier. It will get
incremented by one for every operation requested by application. So unique identifier is clientID
+ operationID
        operationID = 0

        # Config variable which stores current active configuration. It includes the sequence
number of the configuration and an ordered list of replica identifiers.
        configuration config = {}

        # Contact olympus to fetch active configuration and will await for the result
        def getConfigFromOlympus():
                send(('get_config',), to=olympus)
                Await till get a response from olympus
                if response:
                        # Assigned fetched configuration from olympus to global variable config.
                        config = response.config
                End if
------------------------------------------------------------------------------------------------------------

        # Function will iterate over config and will return address of head replica
        def getHeadReplicaFromConfig():
                return config.head;
------------------------------------------------------------------------------------------------------------

        # Get a list of replica from current config. We will be needing this during retransmission
of request.
        def getListOfReplicaFromConfig():
                return config.getReplicaList;
------------------------------------------------------------------------------------------------------------
```

```
# Get a value of t from 2t + 1. This variable is required to get the count of t+1 replica
def getFaultTalurenceCountFromConfig():
        return config.t
```

```
# Get request from application.
def getOperationRequestFromApplication(operation):
        # Increment the global counter as per request
        operationID++

        # Create a unique identifier per request.
        # In order for a replica to determine whether it has a cached result shuttle
        corresponding to a (retransmitted) client request, it needs to be able to associate
        them with each other.
        # This can be done by client by including a unique identifier in each request, and
        for replicas to include this identifier in the result shuttle.

        operationIdentifier = clientID + operationID

        # Initialize the global config variable
        getConfigFromOlympus()

        # Fetch head replica from configuration.
        headReplica = getHeadReplicaFromConfig();

        # Start timer
        startTimer()

        # Send request to head with parameters - operation ,unique operation identifier
        and isRetransmission flag to differentiate between retransmission and normal
        request

        isRetransmission = false
        send((operation, operationIdentifier,  isRetransmission), to=headReplica)
        Await till get a response or timeout
        if response:
                if response == error:
                        # if client receives error message from tail. This is an edge case
                        where our tail is immutable and it will send error message to client.
                        So, fetch new configuration from olympus.
                        getConfigFromOlympus()
                        getOperationRequestFromApplication(operation)
                else :
                        # Need to validate response which consists of actual result and
                        result proof
                        validateResponse(response)
            else if timeout:
                # Client does not receive result from tail. So it will retransmit the the
                request to all replica.
                retransmitRequest(operation, operationIdentifier)
```

# Response is made of result (actual value) and result_proof from replica. In this function we are going to validate computed hash of result from client against at least t+1 result statements of a result proof.

```
 def validateResponse(response):
        # Fetch actual result from response
        result = response.result

        # Fetch Result proof from Response.
        resultProof = response.resultProof

        # Compute cryptographic hash SHA256 of actual result. H(result) denotes crypto hash
function
        hashResult = H(result)

        count = 0
        t = config.get_fault_talurence()
        # Iterate over each result statement to get result state of format - <result, o, S(r)>  from
        resultProof to get signed result - S(r).
        for each resultStatement in resultProof:
                # First we need to decrypt this statement using client's private keys
                # S(r) part of resultStatement
                signedResult = resultStatement.signedResult
                if (signedResult == hashResult):
                        count++;
                if count == t+1:
                        # client received valid hash and send this result to application
                        send result r back to application
        end for
        if count < t+1 :
        # Issue a proof of misbehaviour
                send(('proofOfMisbehaviour', result, resultProof), to=olympus)
        End if
```

    # After timeout, client will send request to all replica with same operation and unique identifier and flag as true because it is retransmission request.

```
  def retransmitRequest(operation, operationIdentifier):
        # Get a list of replica from configuration
        listOfRepilca = getListOfReplicaFromConfig()

        startTimer()
        # Send request to all replica with parameters - operation ,unique operation identifier and isRetransmission flag as true to differentiate between retransmission and normal request
        isRetransmission = true

        for each replica in listOfReplica:
              send((operation, operationIdentifier, isRetransmission), to=replica)
        end for.

        Await till get single response or timeout
        If response :
              result = response.result
              # To avoid redundancy , we will go ahead with 1st result from any replica.
              if result == error:
                    # If any replica is immutable, get new configuration from olympus.
                    getConfigFromOlympus()
                    # Start same operation with new configuration.
                    getOperationRequestFromApplication(operation)
              else :
                    # Need to validate for proof of misbehaviour
                    validateResponse(response);
        If timeout :
              # if retransmission is time out, send again normal request.
              getOperationRequestFromApplication(operation)
```

## Olympus -

# Implements oracle
# Generates series of configurations -> Issue inithist statements signed by private keys
# Configurations C1 and C2 are disjoint if there does not exist a replica that is in both C1 and C2.
# Each configuration keeps list of quorum.
# Each quorum keeps list of replica (at least t+1 replica)
# Olympus creates all public and private keys for client and replica and send them to relevant process. These keys can be generated using Curve25519.
# Olympus has information about inter- replica keys i.e. replica pair keys
# listOfConfigurations is set of configuration. Only one configuration in this set is active. All other configurations are inactive.

```
listOfConfigurations = []

# activeConfig will contain current active configuration.It will contain the list of active replica
activeConfig = {}
```

# 'statements' is an abstract global variable. It is a list of all statements from all replicas. It contains following types of statement -

1 . <**order**, s, o> - This is order statements with slot number and o is tuple of operationType and operationIdentifier

2. <**wedged**, rho.history> - It is wedge statement. rho.history is set of order proofs of respective replica rho.

3. <**inithist**, C , hi> - It is inithist statement. seed history(h) is maximum order proof to new configuration C. At most one inithist can be issued for a configuration.

4. <**result**, o, S(r)> - This is result statements with operation tuple o - operationType and operationIdentifier and S(r) - cryptographic hash of result.
and o is tuple of operationType and operationIdentifier

```
statements = []

# Client requests olympus for active configuration.
def receiveMsg('get_config'):
    return activeConfig
```
--

```
# Reconfiguration_request from replica
def receiveMsg('reconfiguration_request'):
          # called switched config
           switchConfig();
------------------------------------------------------------------------------------------------------------------------------
```

```
# proof_of_misbehaviour request from client. We need to check the result statement received
from client with the abstracts statements in olympus for checking misbehaviour.
def receive_msg('proof_of_misbehaviour', result, result_proof):
          # create hash of result
          hash_result = computer_hash(result)
          count = 0
          for each result_state in resultProof :
                    # First we need to decrypt this statement using tail's private keys.
                    If statement contains result_state:
                              # S(r) part of result state
                              signedResult = result_state.signedResult
                              if (signedResult == hash_result):
                                        count++
                              if count == t+1:
                                        # Client request of proof_of_misbehaviour is not valid
because there are atlest t+1 replica with same hash of result. But we need to check
honesty of result statement with global statements.
                                        For each result_statement in result_proof:
                                                  If result_statement not in statement:
                                                            swicthed_config()
                                        End for
                                        break
                    else
                              swicthed_config()
          end for
          If count < t + 1 :
                    # Client request of proof_of_misbehaviour is valid
                    swicthed_config();
------------------------------------------------------------------------------------------------------------------------------
```

```
def switchConfig():
          # Send wedge request to all replica
          # It is a map of replica ID and replica history
          listOfWedgedStatement = {}

          # Get a list of all replica from active_config
```

```
    listOfRepilca = activeConfig.getList()

    # send request to all replica with wedge-request
    for each replica in listOfReplica:
        send((‘wegde_request’), to=replica)

    Await till we get t +1 responses.
    # First we will decrypt wedged statement using Curve25519.
    # Add all result (received wedged statement) to map.
    listOfWedgedStatement[response.replicaId] = response.history

    # Get quorum containing replica with maximum order proof
    # This function performs 3 operations
    #  1. Consistency check
    #  2. Honesty check
    #  3. Catch_up check
    #  Returns quorum and cryptographic hash of running state of maximum order
proof replica

    quorum,ch = checkConsistency(listOfWedgedStatement)
    replica = verifyRunningState(quorum, ch)

    # get new configuration C’ which is the successor of the current configuration by calling a
function succ(C,C’) where C is the active configuration and C’ is the successor

    # Check statements doesn’t have entry of <inithist, C’, h>. Because the olympus can
issue at most one inithist statement per configuration

    For each statement in statements
        If statement ==  <inithist, C’, h>
        # get a new configuration which doesn’t have inithist entry in statements.
        End if
    End for

    # Now call inithist function for new configuration with selected replica and add it
to statements.
    statements.add(generateInithistStatement(C’, replica.history))
```

```
def generateInithistStatement(C', history):
        # This function will add history to all replica in new configuration and will return
generated inithist statement signed by private key using Curve256 which will get added to
statements.
        return {'inithist', C', history}
-----------------------------------------------------------------------------------------------------------------------

# At least one replica will return running state S whose hash value is same as that of ch that we
have already computed. Olympus includes S in its inithist message as the initial running state of
the new configuration.
def verifyRunningState(quorum, ch):
        for each replica in quorum:
                send('get_running_state'),to=replica)
                Await till get response
                If response:
                        #now compute hash of response and compare it with previously
computed hash ch by client. H(response) denotes the crypto hash function done by SHA256
                        if(H(response) == ch)
                                return replica
        end for
-----------------------------------------------------------------------------------------------------------------------

# check consistency of wedge_statement.
def checkConsistency(listOfWedgedStatement):
        # Get quorum list from current active configuration
        quorumList = activeConfig.quoromList

        # This variable will have final quorum with all consistent and honest wedged statement.
        consistentQurorum = []

        Bool isConsistent = false
        Bool isHonest = false
        Bool isCached = false

        # maximumOrderProofReplica will have a history which will be fed to all replicas of new
configuration.
        maximumOrderProofReplica = {}

        # Checking the consistency of each quorum
        for each quorum in quorumList:
                # Check the consistency of quorum.
                # This can be calculated by checking two replicas history at a time and checking
                #  the order proof.This ensures that two replicas history can not have same slot
                # number for different operation identifier.
                # Consistent" means that, for each pair of replicas rho1 and rho2 in quorum, for
                each slot s for which an order proof appears in rho1.history and rho2.history, the
                order proofs for s are consistent, i.e., are for the same operation.
                # All replicas in quorum have same slot number for particular operation.
                # if the histories are not consistent, then one of the replicas in selected quorum is
                faulty, and Olympus needs to choose a different quorum.

                isConsistent = checkConsistencyOfQuorum(quorum, listOfWedgedStatement)
                If isConsistent == true:
                        #Now we need to check the honesty of all wedge statement for selected
quorum.
                        isHonest = checkHonestyOfQuorum(quorum, listOfWedgedStatement)
                        if isHonest == true:
                                maximumOrderProofReplica =
getMaximumOrderProofReplica(quorum)
                                isCached, ch = catchUp(quorum, maximumOrderProofReplica,
listOfWedgedStatement)
                                If isCached == true:
                                        consistentQurorum = quorum
                        End if
                End if
        End for
return consitentQurorum, ch
```

```
# This is required to check whether the provided quorum is consistent or not .
def checkConsistencyOfQuorum(quorum, listOfWedgeStatement):
        # First retrieve the wedged statement for all replicas in quorum from map
        # Select two wedged statement/ history at a time and check for the consistency of slot
number and operation.
        # If found any discrepancy return false
        # else return true
```

```
def getMaximumOrderProofReplica(quorum):
# It returns the replica having the longest history
        maxLen = 0
        maximumOrderProofReplica = {}
        For each replica in quorum:
                if maxLen < replica.history().length() :
                        maxLen = replica.history().length()
                        maximumOrderProofReplica = replica
                End if
        End for
        return  maximumOrderProofReplica
```

```
def catchUp(quorum, maximumOrderProofReplica , listOfWedgedStatement) :
    # Initialize the cryptographic hash as empty string
    ch = “”
    # Iterate over selected quorum
    For each replica  in quorum:
        If replica != maximumOrderProofReplica :
            # Calculate the catchUpData
            catchUpData = maximumOrderProofReplica.history() - listOfWedgedStatement[replica]
            send((“catup_messgae”, catchUpData ) to = replica)
            Await till get response
            If response:
                if(ch == “”)
                    ch = response.hash
                Else if (ch != response.hash )
                    # we need to select new quorum
                    return false
            end if
    end for
    # every hash is matched, we can go with this quorum
    Return true,ch
```

```
# Now we need to check for honesty of all wedge statement for selected quorum.
def checkHonestyOfQuorum(quorum, listOfWedgeStatement):
    #listOfWedgeStatement is a map of wedged statements containing histories (set of order proof) of each replica.
    For each replica in quorum
        #  Retrieve history of replica from listOfWedgeStatement. Let’s call this as replicaWedgeHistory = listOfWedgeStatement[replica]
        #  Retrieve all order statement from replicaWedgeHistory
        #  Retrieve all order statement  for selected replica and all its previous replicas in current configuration, from statements
        # Now check whether these order statements are present in replicaWedgeHistory or not
        # if found any discrepancy, return false.
        # else return true.
```

Replica :

class Replica(process):

```
# Replica id
rhoID

# mode of replica
# It can be any of these {PENDING, ACTIVE, IMMUTABLE}
mode

# Current active configuration
Configuration C

# Set of order proofs.
# This will get added to global history variable.
history = []

# Keeps info of previous and next replicas
Replica previousReplica
Replica nextReplica

# It is a local cache with the completed proofs for operation. And new value is added to it after tail sends resultShuttle in reverse order.
resultShuttleCache={}

# This variable gets loaded every time when the previous replica sends shuttle.
shuttle ={}

# slot Number
slotNumber = 0
```

def getRhoID():

```
    return rhoID
```

---

def getUniqueSLotNumber():

```
    slotNumber++
    return slotNumber
```

```
def receiveMsgFromOlympus('wegde_request'):
    # Send tuple of history and replica Id to olympus.
    # Handles wedge request from olympus and responds with wedge Statement
    becomeImmutable()
    return <history, rhoID>
------------------------------------------------------------------------------------------------------------------------

def receiveMsgFromOlympus('catchup_messgae', catchUpData):
    # Upon receiving this message, replica executes catchUpData operations and then computes and sends a cryptographic hash of its resulting running state to Olympus in a "caught_up" message.
    return hash(running_state)
------------------------------------------------------------------------------------------------------------------------

def receiveMsgFromOlympus('get_running_state'):
    # Return current running state of replica to olympus.
    return runnig_state;
------------------------------------------------------------------------------------------------------------------------

def becomeActive():
# Pending replica becomes active if olympus issues inithist statement for C
    if mode == PENDING and statements.contain(inithistStatement)
        mode = ACTIVE
------------------------------------------------------------------------------------------------------------------------

def becomeImmutable():
# Happens only if one or more of active replica's peer replicas are suspected of being faulty
    if mode == ACTIVE
        mode = IMMUTABLE
        wedgeStatement = signWedgedStatement(mode,history)
        # olympus figures out the mode as immutable and history from this wedged statement
        statements.add(wedgeStatement )
------------------------------------------------------------------------------------------------------------------------

# Receive shuttle event from other replica.
def receiveShuttleMsgFromReplica(slotNumber, operaion, shuttle, isResultShuttle):
    if(isResultShuttle)
        # It will contain the result and resultProof
        resultShuttleCache.append(shuttle)
        # if completed checkpointproof is present, delete all history of replica before checkpoint. This will get used as part of wegded statement send to olympus.
        If (previousReplica)
```

```
                send(lotNumber, operaion, shuttle,isResultShuttle), to=previousReplica)
    Else
    # Else if it’s a normal shuttle and this is not a head replica, then
    # Identifying head replica using previousReplica
        If (!previousReplica) :

        # Assign shuttle from other replica to current shuttle.
        shuttle = shuttle

        # Applies operation to its running state and obtains a result r.
            # Now compute cryptographic hash of result. H(r) denotes the crypto hash
function using SHA256)
            hashValueOfResult = H(r)

            # call orderCommand to create order proof
            orderProof = orderCommand(C,slotNumber, o)

            # Need to generate resultProof
            # Each result statement inside result proof is signed with public key.
            resultProof = createResultProof(C,slotNumber,o,S(r))

            #Appends shuttle with order proof and result proof
            shuttle.add(orderProof)
            shuttle.add(resultProof)

            # if checkpointproof is present in shuttle
            # Create a checkpoint proof - hash of running state to shuttle and
        send it to next replica.
            shuttle.add(checkpointProof)

            If (nextReplica) :
                # forwarding operation tuple, slot number and shuttle to next
replica if this is not a tail Replica
                isResultShuttle = false
                send(slotNumber, o, shuttle, isResultShuttle), to=nextReplica)
            Else :
                # if it is tail replica, send result and resultProof to client.
                send((result, resultProof), to=client)
                #since this is tail, therefore the shuttle is referred as resultShuttle
            and we have denoted this by isResultShuttle flag being true and then
            send it to previous shuttle
                # It will also send completed checkpoint proof to previous
            replica.
```

```
                        isResultShuttle = true
                        send(slotNumber,     operaion,     shuttle,     isResultShuttle)
                    to=previousReplica
```

```
# Receive message from other replica due to retransmission.
```
def receiveMsgFromReplica(operation, operationIdentifier, isRetransmission):
```
    if(!previousReplica)
        # If it is a head replica.
        receiveMsgFromClient(operation, operationIdentifier, isRetransmission)
```

```
# This function takes care of the actions to be performed on receiving message from client
# Client can send a normal request or retransmission on time out.
# There are 4 cases, which are explained below.
```
def receiveMsgFromClient(operation, operationIdentifier, isRetransmission):
```
    # create operation tuple for action to be performed.
    # o  is <operation, operationIdentifier>
    o = createOperationTuple(operation, operationIdentifier)

    #Normal transmission
    If (!isRetransmission)  :
        # check whether current replica is head or not by checking previousReplica.

        if(!previousReplica):
            # Current replica is head replica
            # Clear shuttle object for new operation with normal request.
            shuttle ={}
            # Generate a slot number
            slotNumber = getUniqueSlotNumber()

            # Applies operation to its running state and obtains a result r.
            # Now compute cryptographic hash of result
            hashValueOfResult = H(r)

            # call orderCommand to create order proof
            orderProof = orderCommand(C,slotNumber, o)

            # Need to generate resultProof
            #  Each result statement inside result proof is signed with public key.

            resultProof = createResultProof(C,slotNumber,o,S(r))
```

```
                    #Appends shuttle with order proof and result proof
                    shuttle.add(orderProof)
                    shuttle.add(resultProof)

                    # create a background thread for checkpoint which will add
checkpoint proof - hash of running state to shuttle and send it to next replica
                    # Let's say checkpoint is done for 100 slot number
                    shuttle.add(checkPointProof)

                    # forwarding operation tuple, slot number and shuttle to next replica
                    isResultShuttle = false
                    send(slotNumber, o, shuttle, isResultShuttle), to=nextReplica)
    Else:
    # when retransmissionFlag is true, it means the message is a retransmission request
from client. There would be different cases which are mentioned below -

    # Case1: if resultShuttleCache of current replica contains shuttle for the operation
o, send the cached result and resultProof  to the client.
    for resultShuttle in resultShuttleCache:
        If resultShuttle corresponds to operation o:
            send((result, resultProof), to=client)
        End if
    End for

    # Case2: if mode is immutable for current replica, then sends error statement to
    client
    if mode == IMMUTABLE:
        send((error), to=client)

    if(!previousReplica):
    # Case3:if this is headReplica , then there can be two cases:-
    # Case3.a: if operation is unknown in the history, head starts the operation from
    scratch and starts timer.
        if !history.contains(operation):
            # Current replica is head replica
            # Clear shuttle object for new operation with normal request.
            shuttle ={}

            # Generate slot number
            slotNumber = getUniqueSLotNumber()

            # Applies operation to its running state and obtains a result r.
```

```
                # Now compute cryptographic hash of result
                hashValueOfResult = H(r)

                # call orderCommand to create order proof
                orderProof = orderCommand(C,slotNumber, o)

                # Need to generate resultProof
                # Each result statement inside result proof is signed with public key.

                resultProof = createResultProof(C,slotNumber,o,S(r))

                # creates shuttle with order proof and result proof
                shuttle.add(orderProof)
                shuttle.add(resultProof)

                # Forwarding operation tuple, slot number and shuttle to next replica
                send(slotNumber, o, shuttle), to= nextReplica)
        Else:
        #Case3.b: the operation is present in the history of the head replica and it's
waiting for the result shuttle
        startTimer()
        Awaits for resultShuttle or timeout:
        if resultShuttle :
                # resultShuttleCache updated for current replica and it contains final
result of operation. So recursively call this function with transmission flag as true.So next
time when this function will get called, Case1 is called and result will be sent.
                receiveMsgFromClient(operation, operationIdentifier, true)
        Else If timeout:
                # On time out send reconfiguration_request message to olympus.
                send((‘reconfiguration_request’), to=olympus)
#Case 4 : if this is not head, then in rest of the cases , sends retransmit request to
head and starts timer
        #identifying head replica by previous replica
        if(previousReplica):
                send((’operation, operationIdentifier, true), to=head)
                startTimer()
                awaits for resultShuttle or timeout:
                if resultShuttle :
                for resultShuttle in resultShuttleCache:
                        If resultShuttle corresponds to operation o:
                                send((result, resultProof), to=client)
                        End if
                End for
```

```
                    Else If timeout:
                        # On time out send reconfiguration_request message to olympus.
                        send((‘reconfiguration_request’), to=olympus)
```

# orderCommand issues the order command in the current replica with the slot number s and operation tuple o.
# After all preconditions are satisfied, order_commmand will create a new order proof

```
def orderCommand(C,s,o):
    # checking the preconditions before issuing a new order statement:-
    # if current replica is active.
    if mode == ACTIVE :
        # Get a list of previous replica from configuration C
        listOfPreviousReplica = C.getListOfPreviousReplica()

        for each replica in listOfPreviousReplica:
            # Get orderStatement of slot number s and operation o of a replica
            If orderStatement is not present in statements:
                # Issue reconfiguration_request to olympus
                send((‘reconfiguration_request’), to=olympus)

            else
            # check whether slotNumber in orderStatements of previous replica’s history matches with the slot number with other operation or not.
            # If yes, reissue configuration because we can not have same slot number for two different operations.
                for each orderProof in replica.history:
                    if( orderProof.slotNumber == s)
                        if( orderProof.operation != o)
                            # Issue reconfiguration_request to olympus
                            send((‘reconfiguration_request’), to= to=olympus)
                        End if
                    End if
                End for
        End for
    End if

    # After checking the above preconditions, it generates signed order statement by using public key of next replica, then generates orderProof and add orderProof to history. Also , it adds orderStatement to the global variable statements.

        orderStatement = generateOrderStatementSignedForCurrentReplica(s,o)
        orderProof = generateOrderProofForCurrentReplica(orderStatement)
        history.append(orderProof)
        statements.append(orderStatement)
        Return orderProof
```
