Adarsh Kashyap, 110782182
Kanishta Agarwal, 110931724

CSE535 –Asynchronous Systems – Fall 17
Project pseudo code

Question : Pseudo code for Bizantine Chain Replication

We present below the idea and pseudo code for implementation of Byzantine-tolerant State Machine Replication protocols for asynchronous environment. Our implementation consists of Olympus, Replicas and Clients class instances which works on majority based quorums and a chain of configured replicas. Replicas in case of failure and misbehavior will be spawned again and reconfigured by the Olympus. We have made a few assumptions while writing pseudo-code. The possible case of them getting violated will be handled in the main code while implementation. One such assumption we make here is that client will always send one of the four possible valid operations. We also have not given the implement details of some of the validity and generating functions like generateId(), validHistory() and others. These functions will be implemented during the actual implementation across different phases.

Support Classes

```

# Data class that will be used by clients to send
# key and values for various operations

class Data:
  key     # Integer mostly
  value     # May be a list since we need to slice and append

# This class is used by client to send a request
# @operationType  - get/put/slice/append oeprations
# @msgId      - a unique messageID specific to this operation
# @data       - data for the operation sepecified
# @clientID     - ID of the client, replicas or head use this to verify
#           if client is registered

class operation:
  def __init__(self,operationType, msgId, data, clientId):
    self.operationType = operationType
    self.msgId  = msgId     # Varchar
    self.data = Data
    self.clientId = clientId  # Varchar/Long

# This class is used to store client details in the cache
# @isResponseReceived - used by timer callback to check if response is
   received
# @operation  - Instance of the operation which is sent to head
# @timerObject  - we save timer object for every request, since we might
   have to
```

```
#           delete the timer when we receive response or when it timesout we need
#         to start new timer.

class ClientMessage:
  def __init__(isResponseReceived, operation):
    self.isResponseReceived = isResponseReceived
    self.operation = operation
    self.timerObject = None

# This is the configuration that client receives during startup from Olympus
# @configID - ID of the config that is active with olympus currently
# @headId   - ID of the head which is currently active in the config
# @tailId   - ID of the tail that is currently active
# replicas  - Dictionary that has all the replicaID and its publickeys
#       [In implementation this will mostly be processID to which we
#       send message]

class clientConfiguration:
  configId  # Long or VarChar unique ID
  replicas = dict(replicaId, keys)
  headId    # Mostly it will be the processID or someunique ID to contact head
  taildId   # ProcessID of the tail or some unique identifier


# Replica config that will be given to the replica during the inialization
# of a replica. It contains all the information such as head info, publickeys of all
# the replicas, previous and next replica ID for every replica, clientID and public keys
# Instance of olympus and more.

class replicaConfig:
  replicaId = None  # VarChar mostly
  configId = None   # VarChar mostly
  headId = None   # ID of the head
  replicas = dict(replicaId, keys) # Map of replicaID and its public key
  stateObj = None
  prevReplicaId = None # ID of the prev replica in the chain
  nextReplicaId = None # ID of the next replica in the chain
  clients = dict(clientId, keys)  #Dict that contains clientID and publickeys
  isHead = False  # Boolean indicating if it is a head
  isTail = False  # Boolean indicating if a replica is tail
  OlympusID = None # ID of the olympus which we will be used to communicate
  dataStore = none # dataStore on which all operations happen
  mode = "ACTIVE" # Mode as described in paper

# This object will be used to create a quorum
# @ quorumId - unique ID per quorum
```

```
# @ replicaList - List of replicas that are part of this quorum
class Quorum:
  quorumId
  replicaList = list()

# This response object is the type which will be used to send the response
# to the client, we include result, a proof for result and messageID.
# messageID will be used by the client to understand which response is received.
class responseObject:
  msgId = None
  result = None
  resultProof = None

# This class is used to bundle up all required information for wedged
# request from olympus
# @replicaID - ID of the replica which sends the wedged data
# @history - history of the replica
# @checkPoint - list of checkpoint information which is held by each replica
# @dataStore - State of the datastore just before sending the wedged object
# This entire wedged objecy will be encrypted using replica private key

class wegded(object):
  replicaId = None
  history = None
  checkPoint = list()

# This is the orderProof shuttle which will be propagated down the chain
# Contains all the required information required for shuttle as per the paper
# It also contains a list of orderstatement to which each replica will append
# the order statement when shuttle is moving down the chain

class OrderShuttle:
   slotNumber
   operation
   replicaId
   configId
   orderStatement = list()

# Order statement that each replica will append to the orderproofshuttle.
class OrderStatement:
  slotNumber
  operation
  replicaId

# Result statement that each replica will append to the result shuttle.
class ResultStatement:
  operation
  crpytHash(Result)
```

```

# Result shuttle that propagates down the chain.
class ResultShuttle:
  list(ResultStatement)

# Checkpoint shuttle that runs down the chain, since we are deciding
# checkpoint period based on number of slots seen, we include the checkpoint slot
# number and hash of the datastore to the shuttle.
# Hash of datastore will be used to check if all of them are at the right checkpoint
class checkPointShuttle:
  slotNumber # Slotnumber
  state = list[hash(dataStore)] # state of the data
```

Client

```
# Main client class
# - 'generateKeys()' will generate public and private key pair.
# - olympus instance will be passed from the driver program.
# - According to our logic, all clients register to the olympus before they
#   start requesting services and they share their keys as well

class Client:
  def __init__(olympusId):
    self.id           # VarChar
    self.publicKey, self.privateKey = self.generatKeys()
    self.config
    self.olympus = olympusId
    self.timeOutCheck = dict(messageId, ClientMessage)
    self.isConfigReady = False  # Boolean
    self.register()

  # Starts a timer in a seperate thread this will periodically check
  # if the olympus configuration is changed.
  def startGlobalTimer(time, olympusConfigCheck):
    threading.Timer(time, olympusConfigCheck).start()

  # This method is a callback for timer to verify is config is changed
  # - If config is changed in olympus then for every request that we have not received
  # response, we retransmit and we also stop their old timers.
  def receive("configChanged", olympusId, isChanged):
    if isChanged:
      self.fetchClientConfig()
      for id, operation, timerObject in timeOutCheck.items():
        timerObject.stop
        del timeoutCheck[id]
        self.sendRequests(operation, "retransmission")

  def olympusConfigCheck():
    send("isConfigChanged", clientId, configId, to=olympus)

```

```

# This method will be invoked in the begining to fetch configuration from olympus.
def fetchClientConfig():
  send("getClientConfig", to=olympusId)

def receive("clientConfig", clientConfig, olympusId):
  if valid(olympusId):
    self.clientConfig = clientConfig

# This method would generate public and private key for the client.
def generatePublicKey():
  pass

# This API will be used by client during the initialization, where the client registers
# itself with the Olympus with its public key.
# While registering we pass callback function which will receive the config from olympus.
def register():
  # do inititialization, setup connection
  send("registerClient",self.id, self.publicKey, to= olympusId)

# Setter for configuration
def setConfig(config):
  self.config=config

# callback function where configuration is received
# we plan to use inConfigReady boolean to check if configuration is received
# before doing multiple operations.
def receiveConfig(config):
  self.setConfig(config)
  self.isConfigReady = True
  self.startGlobalTimer(time, self.olympusConfigCheck)

# This is a verification method which will verify that the result
# matches for atleast t+1 replicas.
def verifyResult(responseObject):
  if (verification fails):
    self.reportMisbehaviour(responseObject)
    return False
  else
    return True

# This method is invoked when client suspects something is not right in the result
# This method tells olympus to reConfigure itself, and we have a callback to which
# olympus responds once the reconfiguration is finished.
def reportMisbehaviour(responseObject):
  # May have to send result proof
```

```
    send("reConfigure",self.id, responseObject)

  # This method is used to make a request to perform any action
  # operationType - get/put/slice/append operations
  # data - key / key and value
  # 'generateID' method generates a unique ID for the request
  def makeRequest(data, operationType):
    msgId = generateID()
    operation = self.createRequestObject(operationType, msgId, data, self.
  id);
    sendRequests(operation, "new")

  # operation - Operation object with type, data and other details
  # messageType - Possible values :
  #           "New" - new request
  #         "retransmission" - retransmission

  def sendRequests(operation, messageType):
    # data is a tuple that will have values depending on the type.
    # Will be key value pair or just value
    obj = ClientMessage(False, operation)
    self.timeOutCheck.add(operation.id, obj)
    sendMessage(operation.clientId, self.encrypt(operation), messageType);
    timer = startTimer(timeout=config.timeout, callback=self.timerCallback,
    args=id)
    obj.timerObject = timer

  # This method sends the message to the head incase of a new request
  # or broadcasts to all replicas in case of retransmission.
  # All the replica details are obtained as a part of client configuration.

  def sendMessage(clientID, operation, messageType):
    if (messageType=="new"):
      self.send(clientID, operation, to=config.headId)
    else
      self.broadcast(clientID, operation, to=confi.replicas)

  # This is a handler for receiving message of type 'client'.
  # This message is basically a response for some request and
  # we identify the message by using ID which is a part of responseObject

  def receiveResponse("client", responseObject):
    msgId = resposeObject.msgId
    if(verifyResult(resposeObject))
      timerObject = self.timeOutCheck[msgId].timerObject
      timerObject.stop
      del timerObject[msgId]

  # This method encrypts any object using the client public key
  def encrypt(object):
    pass
```

```
  # This method is used to generate a unique clientID
  # Note: we have different clientID and messageID
  def generateID():
    pass

  # This is a callback function for message timers, if any message timer
  # timesout we delete the old timer for that message and retransmit the
  # request again.
  def timerCallback(uid):
    if(timeOutCheck[uid].isResponseReceived == False):
      operation = timeOutCheck[uid].operation
      del timeOutCheck[id]
      sendRequest(operation, "retransmission");
```

Olympus

```

# This is the main olympus class
# we have initialized all important datastrucuters here.
# Some of these will take the value from the driving program

class Olympus:
  def __init__():
    self.clientConfiguration = None
    clients = dict(clientId, clientKey)
    replicas = dict(replicaId, replicaKey)
    replicaList = []
    configs = list()
    activeConfig
    QuorumList = list(Quorum)
    replicaWedged = dict(replicaId, wedged)
    dataStore = dict()
    replicaDataHash = dict(replicaId,dataStoreHash)
    head = None
    isQuorunReceived = False
    numReplicas #
    __privateKey = None
    makeConfig()

  # This methid generates a new ID for configuration, spawning and adding
   the
  # config ID to the list of configurations.

  def makeConfig():
    # We have to check succ(oldConfig, newConfig) if required here
    configId = generateConfigID()
    spawnReplicas()
    activeConfig = configId
    configs.add(configId)

  # This method will make the object of type replicaConfig, we have not
   shown
  # populating the arguments, please refer replicaConfig class structure.
```

```
def makeReplicaConfig():
  return replicaConfig()

# This is the main method that spawns all the processes. We first create multiple
# process and we get their ID's and before we start them, we update all the info
# required in the setup i.e. we add details such as ID of the next and prev process
# which is a part of replicaConfig object.
# 'generateKey' is a method which generates public and primary key.
# This also takes care of deleting old processes if this is not the first time we are
# making the configrations. We check this by checking the size of a list that holds
# complete data of all processes spawned.

def spawnReplicas():
  if replicaList is empty:
    for i in range(numReplicas):
      replicaList[i] = new process("replica.class")

    head = replicaList[0];
    for process in replicaList:
      # Make the replica config object with all the details required for it.
      replicaConfig = makeReplicaConfig()
      setup(replicaConfig)
      process.start()
      privateKey,publicKey = generateKeys();
      replicas[process] = publicKey
  else:
    for i in range(numReplicas):
      deleteProcess(replicaList[i])
      spawnReplicas()

# Using numReplicas, we will generate all combination of t+1 quorum, where
# each quorum is a tuple of replicaID's. This also generates a unique quorum ID
def populateQuorum():
  pass

# This method checks if a configuration is succesor of another.
# we have maintained a list of configurations which will be used here.
def succ(oldConfig, NewConfig):
  pass

# This method makes client configuration object, it adds all the required
# information such as Current active configId, dict(replicaId, keys), headId and tailId
# Client uses this info while starting up
```

```
def makeClientConfig()
  return clientConfiguration()

# This handler is used to register client during the initalization
# of the client, we save the public key of the client with its unique ID
# This ID will be used to verify if client is valid everywhere

def receive("registerClient", clienId, clientKey):
  clients[id] = clientKey
  send("clientConfig", makeClientConfig, to=clientId)

# This receiver handler receives a request from client to reconfigure
# replicas incase if there is some problem with result.
# Client sends a proof by attaching the response object and the method
# 'validateRequest' validates the request.
# If request is valid then we initiate make new config and send a notification
# to client saying that reconfiguration is complete

def receive("reConfigure", clientId, responseObject):
  if (validClient and validateRequest(responseObject)):
    makeConfig()
    send("configChanged", True)

# This receiver hander is used to send latest configruation to the
# client. This will be called if client realizes if reconfig is called on the
# replicas.

def receive("getClientConfig"):
  send("clientConfig", makeClientConfig, to=clientId)

# This is a receiver handler to recieve isConfigChanged request from
# client, this will just check if the current active config is same as the
# the one that client has sent.
# @configID - config ID that client is using currently

def receive("isConfigChanged",configId):
  if (configId != activeConfig) and validClient
    return True
  else
    return False

# Here we verify is the orderProof send by a replica has some issue
def verifyOrderProof(orderProof, replicaID):
  pass

# This method validates history of all the replicas inthe quorums.
# checkConsistency method will use checkpoints and history together to
# check consistency and also returns the longest histroy to which everyone
```

```
# has to catchup.

def validateHistory(replicaList, quorumId):
  wedgedList = []
  for replica in replicaList
    wedgedObj = replicaWedged[replica]
    wedgedList.add(wedgedObj)

  catchHistory = checkConsistency(wedgedList):
  # If it is none, something is wrong with the histroies received
  if (catchHistory == None):
    return False
  else:
    return sendCatchtUpReq(replicaList, catchHistory, quorumId)

# This method will request all the replicas in quorum to catchup and it will send
# the history details which replicas will use to catchup
# We will use timer to check of all of them have received the caught up message from all
# replicas in the quorum and if we don't receive then we neglect this quorum.
def sendCatchUpReq(replicaList, catchHistory, quorumId):
  for replica in replicaList:
    send("catchUp", catchHistory, quorumId, to=replica)

  #verifies wether all have same hashDataStore or not
  if allCaughtup(replicaDataHash):
    self.DataHash = replicaDataHash
    return True

  return False

def receive("caughtup", replicaId, encrptedDataStore):
  replicaDataHash[replicaId] = encrptedDataStore

# This api will check if
def validateQuorum(quorumId):
  return validateHistory(quorum.replicaList, quorumId)

# This starts checking one quorum after another
# PopulateQuorum() dynamically makes quorum as and when we receive more wedged data and also
# it assigns a quorum ID as well, quorum ID could be hash of the quorum as well
# Once valid quorum is identified we send running state request to one of the replicas
def startQuorumCheck():
  # This method will populate quorumList in background and first quorum is the t+1 that we
  # receive in the begining and the quorum is populated as and when we recieve more.
```

```
    # We might have to use hashing on quorum to make sure we do not repeat same quorum
    populateQuorum()
    validQuorum = None

    for quorum in QuorumList:
      res = validateQuorum(quorumId)
      if (res == True):
        validQuorum = quorum
        break   # Found a valid quorum

    # We have found a valid quorum
    if validQuorum:
      send("send_runningState", olympusID, to=replica.random() in validQuorum.replicaList)

  # This receives reconfigure from replicas.
  # We first verify if the replica is proper and also we verify if the request is valid
  # based on the proof attached.
  # We send wedge request to all the replicas if authenticity is proved.
  def receive("reconfig", orderProof, replicaId):
    if (validReplica(replicaID) and verifyOrderProof(orderProof, replicaID)):
      for replica in replicaList:
        send("wedged", to=replica)

  # This is a receiver handler for receiving wedged data from replica, we add all the wedged data to a map
  # After adding, we check if we have received wedged data from all replicas of the quorum and if we have
  # received all then we set a global variable.
  # As soon as we receive t+1 replica wedge, we start checking the quorums.
  # It is more of like this receive keeps receiving and quorum is formed as and when we received

  def receieve("wedged", wedgedObj, replicaID):
    replicaWedged[replicaId] = wedgedObj
    if (len(replicaWedged) == (numReplicas - 1) / 2):
      startQuorumCheck()

  # This is a receiver handler for receiving running state object from a random replica.
  # verifyDataStore - This method hashes the dataStore and verifies if it is same as the hash that
  #          we have saved, and if it is we call make config to create new config
  #          If we fail to verify, we send to someother replica randomnly to fetch runningState
  # @replicaId : replica that is sending the running state
  # @encryptedDataStore : DataStore encrypted with replicas public key

```

```
  def receive("runningState", replicaId, encryptedDataStore):
    if validReplica(replicaId):
      self.dataStore = decrpyt(encryptedDataStore)

    if(verifyDataStore(self.dataStore) True)
      makeConfig()
    else:
      send("send_runningState", olympusID, to=replica.random() in
   validQuorum.replicaList)
```

replica

```
# This is the main replica class that all replicas will use, Replica
   recevies
# an entire set of configurations from the olympus during initialization.
   We have added all
# componenets to replica to understand what a replica is composed of.

class Replica:
   def setUp(replicaConfig):
     olympus = replicaConfig.Olympus
     replicaId = replicaConfig.replicaId
     configId = replicaConfigreplicaConfig.configId
     headId = replicaCoreplicaConfig
     replicas = replicaConfig.replicas
     stateObj = replicaConfigNone.stateObj
     prevReplicaId = replicaConfig.prevReplicaId
     nextReplicaId = replicaConfig.nextReplicaId
     clients = replicaConfig.clients
     isHead =  replicaConfig.isHead
     isTail = replicaConfig.isTail
     mode = replicaConfig.mode   # mode of the replica
     retransmissionTimeOut = replicaConfig.retransmissionTimeOut
     history = None  # history is a list that contains order shuttle for all
     slots
     wegded = None   # wedged object for each replica to send
     resultStatement = None
     orderShuttle = None
     resultShuttle = None
     checkPointShuttle = None
     slot = None   # An Integer slot number which is unique for all messages
     operation = None  # operation supplied by client
     checkPoint    # Latest checkpoint slot number
     dataStore = replicaConfig.datastore # Dictionary on which all
   operations are executed
     cache = dict(operation.msgId, (result, resultShuttle)) # cache to store
     result for messages
     timeOutCheck = dict(operation.msgId, [isMessageReceived, timerObject])

  # This api handles reTransmission when client sends the same messageID
   which replica
  # has already seen. It handles two cases :
  # - If cache of result is present, it responds to the client with result
```

```
# - If result is not there, it will start timer and asks head to take over

def handleReTransmission(operation):
  #Case when entry is not available
  if cache.get(operation.msgId) is not None:
    response = responseObject(operation.msgId, cache[operation.msgID].result,
                cache[operation.msgID].resultShuttle)
    send("client", response, to=operation.clientId)
  #case when replica doesnot have result
  else:
    send("retransmission",operation, to=headId)
    timer = startTimer(timeout=retransmissionTimeOut, callback=timerCallback,
              args=operation.msgId)
    timeOutCheck[operation.msgId] = [False,timer]

# Call back for timers, if timerticks out and there is no result yet then we report
# misbehaviour
def timerCallback(msgId):
  if(timeOutCheck[msgId].isMessageReceived == False):
    reportMisbehavour()
    #handle become immutable and break
  send("client", cache[msgID].result, cache[msgID].resultShuttle)

# - This starts a order and result shuttle. We handled different cases like
#   if the replica is an Head or is it a tail or normal replica. If we reach the tail,
# - we start a new result shuttle in backward direction and also send result to client.
# - 'verifyResultShuttle' and 'verifyOrderShuttle' will take the responsibilites
#   of verifying the order and result proofs.

def startTransmission(operation, slotNumber, orderShuttle, resultShuttle)

  if (self.mode is not ACTIVE):
    break;

  # Case to check if the replica is head
  if self.isHead:
      # Need a lock here probably
      slot = allocateSlot()
      cache[operation.msgId] = None
      result  = executeOperation(operation)
      orderShuttle = makeOrderShuttle(slotNumber,operation)
      resultShuttle = makeResultShuttle(result,operation)
      history.append(orderShuttle)
      # If head has n slots then we start checkpoint shuttle form head.
```

```
      if (len(history) has replicaConfig.checkPointPeriod slots):
        initiateCheckPointShuttle()
      send("order",orderShuttle, to= nextReplicaId)
  else
    slot = slotNumber
    orderShuttle = orderShuttle
    resultShuttle = resultShuttle
    if verifyOrderShuttle(orderShuttle):
      result  = executeOperation(operation)
      orderShuttle.orderStatement.append(makeOrderStatement(slotNumber,
                        operation,replicaId))
      resultShuttle.append(makeResultStatement(result, operation))
      history.append(orderShuttle)
      # If it is tail then we need to start a result shuttle back and also send
      # result to client
      if isTail:
        cache[operation.msgId] = resultShuttle
        send("result", operation, result, resultShuttle, to=prevReplicaId)
        response = responseObject(operation.msgId, result, ResultShuttle)
        send("client", response, to=operation.clientId)
      else:
        send("order",orderShuttle, to= nextReplicaId)
    else
      reportMisbehavour(orderShuttle, operation.replicaId)
      # Handle becoming immutable

# Receive handler to receive all messages of type order, this is a common receiver
# for both order shuttle and result shuttle while going down the chain.

def receive("order", operation, slotNumber= None, orderShuttle = None,
                           resultShuttle = None):
  if (!verifyClient(operation.clientId) && self.mode == "ACTIVE")
    return
  if (cache.contains(operation.msgId)):
    handleReTransmission(operation)
  else:
    startTransmission(operation, slotNumber, orderShuttle, resultShuttle)

# Receiver handler for result shuttle while the shuttle is moving back in the chain
# Cases : - Is this result shuttle part of normal operation.
#     - If the result shuttle is a part of retransmission, then we cache the result, delete timer and
#      send result to client and move the shuttle up the chain. We need to be careful about the head here.

def receive("result", operation, result, resultShuttle):
  # Checking the mode
  if(self.mode is not valid)
```

```
    return
  # Whatever we have received is from retransmission
  if (timeOutCheck[operation.msgId] != None):
    cache[operation.msgId] = resultShuttle
    timeOutCheck[operation.msgId].timerObject.stop()
    del timeOutCheck[operation.msgId]
    send("client", result, resultShuttle)
    if (!isHead):
      send("result", operation, result, resultShuttle, to = prevReplicaId)
  else:
    cache[operation.msgId] = resultShuttle
    result = result
    if (!isHead):
      send("result", operation, result, resultShuttle, to = prevReplicaId)

# This api starts the shuttle.
def initiateCheckPointShuttle():
    checkPointSlot = getSlotFromHistory()
    self.checkpoint.append(checkPointSlot)
    checkPointShuttle = checkPointShuttle(self.checkPoint, [hash(dataStore)])
    send("checkpoint", checkPointShuttle, to=nextReplicaId)

# Receiver handler for checkpoint shuttle. Here we have seperate cases for
# - Is the shuttle moving forward ?
# - Is the shuttle moving backward ? If so we need to truncate the history
# - Is the replica a tail ? If so we reverse the direction

def receive("checkpoint", type, checkPointShuttle):
  if(self.mode is not valid)
    return
  if (type == "forward")
    self.checkPointShuttle = checkPointShuttle
    if(verifyCheckPoint(self.checkPointShuttle))
      self.checkPointShuttle.state.append(hash(dataStore))
      if (!isTail)
        send("checkPoint", self.checkPointShuttle, "forward", to=nextReplicaId)
      else
        send("checkPoint", self.checkPointShuttle, "backward", to=prevReplicaId)
  elif(type=="backward")
    checkPointSlot = checkPointShuttle.slotNumber
    truncateHistory(checkPointSlot)
    if (!isHead)
      send("checkPoint", self.checkPointShuttle, "backward", to=prevReplicaId)

```

```
# Here basically we go through every data in histry and apply that operation for that
# particular slot to the dataStore of the replica.
# @catchUphistory - history details that we need to catchup
def catchUp(CatchUphistory):
  dataStore.update(catchUpHistory)

# This is a receiver handler for catchUp message from olympus.
# @catchUphistory - history details that we need to catchup
# @olympusID - ID of the olympus
# We return the encrypted hashed dataStore, this will be used by olympus
# to verify that all replicas are in the same state.
#On receiving catchUp request we make an exception to Immutable state of replica and perform catchup opertions

def receive("catchUp", CatchUphistory, quorunId, olympusID):
  if (!verifyOlympus(olympusID)):
    return
  catchUpHistory = decrypt(CatchUphistory)
  catchUp(history)
  send("caughtUp", encryptedHash(dataStore), quorumId, to=self.olympus);

# This receiver handler is used to send the running state of the dataStore when the olympus
# requests for the same. We encrypt the dataStore and send it to the olympus.

def receive("send_runningState", olympusID):
  if (verifyOlympus(olympusID)):
    send("runningState", self.replicaId, encrypt(dataStore), to=self.olympus)

# - Receiver handler for wedged request from olympus, when this request is received
#   we need to make the wedged object and send it to olympus.
# - We make ourself immutable as well just before sending the response
# - We also encrypt the wedged data before sending.

def receive("wedged", olympusID):
  if (verifyOlympus(olympusID)):
    wegdedObj = Wedged(self.replicaId, self.history, self.checkPoint)
    wegdedObj = encrypt(wegdedObj)
    send("wedged", wegdedObj, quorumId, to=self.olympus)
    self.becomeImmutable()

# This method invokes olympus when some misBehavious is identified, it attached
# orderProof in which misbehaviour was identified as a proof.
# We also share replicaID to verify that such requests are received from replicas only.

def reportMisbehavour(orderProof, replicaId):
```

```
    send("reconfig", orderProof, replicaId, to=self.olympus)

  # This will be used only by head when retransmission request is received from any replica
  def receive("retransmission", operation):
    if(self.mode is not valid)
      return
    #Case when entry is not available
    if cache.get(operation.msgId) is not None:
      response = responseObject(operation.msgId, cache[operation.msgID].result,
                          cache[operation.msgID].resultShuttle)
      send("client", response, to=operation.clientId)
    elif (cache[operation.msgId] == None):
      timer = startTimer(timeout=retransmissionTimeOut, callback=timerCallback,
                  args=operation.msgId)
      timerObject = timer
      timeOutCheck[operation.msgId] = [False,timerObject]
    elif (operation.msgId not in cache):
      startTransmission(operation, None, None, None)
```