Team Members:

Pranjal Sahu 110931955
Santiago Vargas 108721656

Data Objects

 # User will create this object to send using a Client
UserRequest {
  fields:
    objectId
    operation
}

 # Client creates this object to send to a head replica
ClientRequest {
  fields:
    objectId
    operation
    clientKey
    requestId

  methods:
    constructor(obj_id, op, key):
      self.objectId = obj_id
      self.operation = op
      self.clientKey = key
}

 # Result received from a tail replica
Result {
  fields:
    requestID
    result
    list resultProof
}

OrderProof {
  fields:
    proofs *# list of proofs*
    issuer *# the issuer of this proof*
}

Proof {
  fields:
    issuer *# the issuer of the proof which will be the replica id*
    operation *# the operation which was performed*
    objectId *# object on which operation was performed*
    slot *# the unique operation number*
}

Classes

 # User will create a Client to send/receive multiple UserRequests
Client {
  fields:
    key
    timeoutSpan
    list sent *# list of sent requests*
    list received *# list of received responses*
    requestCount

  ***method declaration*:**

    sendRequest(UserRequest)

    __sendRequest(UserRequest, msg, destReplicas)

    handleReplicaResponse(Result)

    handleError(Error)

  implementation:
    constructor(tSpan):
      self.key = Olympus.obtainClientKey()
      self.timeoutSpan = tSpan

    # User should use this interface

```
    sendRequest(UserRequest):
        # Get a unique request id
        requestID = getUniqueReuqestID()

        # Get the current configuration
        config = Olympus.getCurrectConfig()

        # Send a request to the head replica
        dest = config.head
        return __sendRequest(UserRequest, msg = "newRequest", destReplicas = dest)


    __sendRequest(UserRequest, msg, destReplicas):
        config = Olympus.getCurrectConfig()

        # Create a request
        request = new ClientRequest(obj_id = UserRequest.objectId, op =
UserRequest.operation, key = self.key)

        # Send the request to the destination replicas
        send sign(<msg, request>) to destReplicas       # Sign with the replica's public key
        sent.add(request)

        # The request has been sent and wait for at max timeout interval for the result
        timeout(minutes = self.timeoutSpan):
            # Check if we have received
            if request.requestID not in received:
                # Get the current config. This updates the config incase there are any changes
                config = Olympus.getCurrectConfig()
                self.__sendRequest(UserRequest, msg = 'retranmission', destReplicas =
config.getAllReplicas())


    # Receive an unvalidated result
    # Note: We assume a user may get multiple responses for a single request
    handleReplicaResponse(Result):
        if Result.requestId in sent:
            for each proof in Result.resultProofs
                replicaPublicKey = config.replicas(proof.replicaID).publicKey
                if SHA256(Result.result, replicaPublicKey) != proof.signature
                    return error

        return Result.result
```

```
        # Received an error. This means the sender replica is immutable
        handleError(Error):
            config = Olympus.getCurrectConfig()
            __sendRequest(UserRequest, replicas = config.getAllReplicas())
}

Olympus {
    fields:
        config                  # Ordered list of replicas
        replicaCount            # The number of replicas
        list wedge_replies      # list of wedge replies obtained from replicas
        list catchUpReplies     # list of catch up replies obtained from replicas
        list runningStateReplies # list of running state replies from replicas
        list publicReplicaKeys  # list of public keys for replica
        list privateReplicaKeys # list of private keys for replicas
        replyTimeout            # acceptable amount of time to wait for a reply from a replica

    methods:

        initHist(numberOfReplicas, config, hist, runningState)

        receiveClientKeyRequest()

        getCurrectConfig()

        checkValidWedgeReply(wedge_reply)

        validateOrderProof(orderProofs)

        handleAsyncRequest(msg, request, client)

        handleWedgeResponse(wedgeResponse)

        handleCatchupResponse(catchupResponse)

        handleRunningStateResponse(runningStateResponse)

        validateCheckpointProof(checkpointProof)

        receiveReconfigureRequest()
```

```
    constructor(numberOfReplicas, timoeut):
        self.replicaCount = numberOfReplicas
        self.config = new Config()
        # Initialize a configuration with empty history and empty running state
        initHist(self.replicaCount, self.config, hist = {}, runningState = {})
        # Generate new public and private keys
        self.publicReplicaKeys, self.privateReplicaKeys = generateSignKeys(replicaCount)
        self.replyTimeout = timeout

    # Creates a new configuration
    initHist(numberOfReplicas, config, hist, runningState):
        # Delete old replicas
        config.purgeReplicas()
        # Generate new public and private keys
        self.publicReplicaKeys, self.privateReplicaKeys = generateSignKeys(replicaCount)

        # Create each replica
        for id in numberOfReplicas:
            r = new Replica(publicKeys = self.publicReplicaKeys, secretSigningKey =
self.privateReplicaKeys[id], status = Active, history = hist, runningState = runningState)
            # Add the replica to the config
            config.add(r)

        # Link each replica
        for each replica in config:
            if replica is not the head:
                replica.previousReplica = replica.index-1
                replica.headReplica = first replica in config
            if replica is not the tail:
                replica.nextReplica = replica.index+1
            replica


    # Returns a new identifier and key to the client
    receiveClientKeyRequest():
            return generateRandomKey()


    # Asynchronous method to handle requests from Clients to obtain the replica list
    getCurrectConfig():
        return self.config
```

```
checkValidWedgeReply(wedge_reply):
   for order_proof in wedge_reply.order_proofs:
        replicaId    = order_proof.issuerId
        signKey     = signKeys[replicaId][replicaId+1]
        if (unsign(signKey.privateKey, order_proof.data) == Success )
           order_proof_data = unsign(signKey.privateKey, order_proof.data)
        else:
           return False

validateOrderProof(orderProofs):
    for order_proof in order_proofs:
      # Check source of order_proof by using replica's public key
      if decrypt(order_proof) == False:
         return False
      # Check if only one distinct operation exists in this order proof
      if order_proof.operations.distinct.count != 1
        return False                                  # Not correct, therefore false
      if order_proof.slots.distinct.count != 1        # Check if only one slot is present
        return False
      return True


# Asynchronous handler for all messages. This multiplexes to sub-handlers
handleAsyncRequest(msg, request, client):
          switch msg:
      'wedgeResponse':
         handleWedgeResponse(request)
      'catchupResponse':
         handleCatchupResponse(request)
      'runningStateResponse':
         handleRunningStateResponse(request):
      'reconfigure':
         receiveReconfigureRequest()
      'configuration':
         getCurrectConfig()
      default:
               return Error




# Handler for wedge responses from replicas
```

```
handleWedgeResponse(wedgeResponse):
    # Append all the wedge responses obtained
    wedge_replies.append(wedgeResponse)


# Handler for catch up messages from replicas
handleCatchupResponse(catchupResponse):
    # Append all the catch up responses obtained
    catchUpReplies.append(catchupResponse)


# Handler for runningState messages from replicas
handleRunningStateResponse(runningStateResponse):
    # Append all the running state responses obtained
    runningStateReplies.append(runningStateResponse)


# All the hashes should match in a checkpointProof
validateCheckpointProof(checkpointProof):
    hash h_hash = {}
    hash h_seq  = {}
    for proof in checkpointProof
        if decrypt(proof) == True:
            h_hash[proof.hash] = 1
    # There is only hash present in all the proofs inside checkpointproof
    if h_hash.keys().size > 1
        return False
    # Sequence number of all the proof in checkPointProof should be same
    if h_seq.keys().size > 1
        return False
    return True


receiveReconfigureRequest():
    # Request wedge statements from each replica
    wedge_replies = []
    for each replica in config:
            # signing uses the private key of the Olympus
            send sign(<"wedgeRequest">) to replica

    # Wait till obtaining t+1 wedge replies from the replica
    # This will create a quorum of t+1 replicas which will ensure that there is at least one
    # honest replica
```

```
While (wedge_replies.size < (replicaCount - 1)/2 + 1):
   True                              # Keep looping


# Find the maximal orderproofs
# slot_op contains the maximum length of order proof for each slot and the operation
        done in each slot
# Create a list of [slot, operation] tuple to recreate replica


hash slot_op = {}                          # Length and operation for each slot
for wedge_reply in wedge_replies:
   # Check valid source of wedge_reply using the public key for that replica
   if (decrypt(wedge_reply) == True)
     # Check valid order_proofs and checkpoint proofs
     if(validateOrderProof(wedge_reply.order_proofs)
          and validateCheckpointProof(wedge_reply.checkpoint_proof)):
         if slot_op[wedge_reply.order_proof.slot] is empty:
           slot_op[slot] =  [wedge_reply.order_proof.operation,
                                          wedge_reply.order_proof.length]
         else if slot_op[slot] is present:
            if slot_op[slot].length < wedge_reply.order_proof.length
               slot_op[slot] = [wedge_reply.order_proof.operation,
                                          wedge_reply.order_proof.length]
            else
               continue
     else
       continue
   else:
     continue


 # Creating history to be used for reconfiguring the replicas
hash h = {}
for slot in slot_op:
   h[slot] = slot_op[slot].operation

for wedge_reply in wedge_replies:
   validateCheckpointProof(wedge_reply.checkpoint_proof)
```

```
        # Create catch_up messages content for checkpoint purpose
        for replica in config: # For each replica
            # Calculate the difference between validated slot operations and the replica's history
            (wedge_reply.order_proofs)
            catchupDifference = wedge_replies[replica.id].order_proofs.keys - slot_op.keys
            # signing uses the private key of the Olympus
            send sign(<"catchUpRequest", catchupDifference>) to replica

        # Wait an acceptable amount of time for catchup responses
        # If a replica does not reply in an acceptable amount of time then they are faulty
        # Therefore, we are guaranteed at least t + 1 responses since t + 1 replicas are correct
        catchUpReplies.purge()                  # Clear the reply list
        timeout(self.replyTimeout):
            # Dictionary of catchup responses (running state hases) with a default value of 0
            hash stateHashes = {}

            # Once the timeout is over, go through the list of response hashes
            for catchUpResponse in self.catchUpReplies:
                # Decrypt the hash using the replica's public key. This validates the replica's identity
                decrypt(catchUpResponse)
                stateHashes[catchUpResponse.stateHash] += 1

            stateHashes.sort()                  # Sort by value

        # Since the list is sorted, the last hash should have the largest and should have a value of
        # at least t + 1 (at least t+1 replicas have the same hash)
        correctHash = stateHash.last().key
        for catchUpResponse in catchUpReplies:          # Already decrypted
            if catchUpResponse.stateHash == correctHash:
                self.runningStateReplies.purge()        # Clear the response list
                send sign(<"runningStateRequest">) to catchUpResponse.replica
                # If we do not receive a response then try another replica
                timeout(self.replyTimeout):
                    for stateResponse in self.runningStateReplies:
                        # Decrypt using the replica's public key and check if the hashes match
                        if decrypt(stateResponse) and  if SHA256(stateResponse.runningState) ==
correctHash:
                            correctRunningState = stateResponse.runningState
                            break(2)            # Break 2 levels - out of the outer loop
```

```
        # Create a new configuration
        initHist(self.replicaCount, self.config, hist = slot_op, runningState = correctRunningState)
}

Replica {
   fields:
      previousReplica
      nextReplica
      headReplica          # the address to head Replica
      history
      interReplicaKeys     # public keys of all replicas
      privateKey           # the private key of this replica
      status
      cache                # cache for storing the results
      headFlag             # boolean flag to check if replica is head or not
      tailFlag             # boolean flag to check if replica is tail or not
      timeoutSpan          # the timeout for retransmission requests
      pendingRequests      # requests whose result shuttle is pending
      allowedOperations = ['query','update' ]
      maxSlotNum           # the largest slot number ordered
      runningState         # contains the values of all objects in the replica
      checkpointNum        # the latest slot number which has been checkpointed
      checkpointHash       # the hash of the running state at the latest checkpoint
      checkPointProof      # latest checkpoint Proof

   methods:
      checkpoint(checkPointSlot)

      handleAsyncRequest(msg, request, client)

      handleNewRequest(client, request)

      handleRetranmissionRequest(client, request)

      handleWedgeRequest(request)

      handleResultShuttle(result, resultProof)

      handleCheckpointShuttle(CheckPointProof)

      handleCheckpointReturnShuttle(CheckPointProof)

      handleCatchupRequest(request)
```

```
handleRunningStateRequest(request)

getProofsFromHistory(required_slot)

orderCommand(order, orderProof, resultProof, sendFlag = True)



constructor(publicReplicaKeys, secretSigningKey, status, history, runningState):
    self.status              = status
    self.interReplicaKey     = publicReplicaKeys
    self.privateKey          = secretSigningKey
    self.maxSlotNum          = max(history)  # Find the maximum slot number in the history
    self.checkpointInterval  = 100          # Denotes how often to checkpoint in terms of new slots ie. every 100 slots
    self.runningState        = runningState


# This function performs a checkpoint up to the chechPointSlot
checkpoint(checkPointSlot):
    # Update checkpointSlot number
    self.checkpointNum = checkPointSlot

    # Hash the running state
    self.checkPointHash = SHA256(self.runningState)     # calculate hash using SHA256


# Catch all message handler
handleAsyncRequest(msg, request, client):
        switch msg:
      'newRequest':
        handleNewRequest(client, request)
      'retransmission':
        handleRetranmissionRequest(client, request)
      'wedgeRequest':
        handleWedgeRequest(request)
      'catchUpRequest':
        handleCatchupRequest(request)
      'runningStateRequest':
        handleRunningStateRequest(request)
      'returnShuttle':
        handleResultShuttle(result, resultProof)
```

```
            'checkpointShuttle':
                handleCheckpointShuttle(CheckPointProof)
            'checkpointReturnShuttle':
                handleCheckpointReturnShuttle(CheckPointProof)
            default:
                    send sign(<"error", request.requestid>) to client       # Sign using replica's
private key


    # Asynchronous handler for normal order command requests
    # Multi purpose handler to handle new requests
    handleNewRequest(client, request):

        if self.state != ACTIVE:                                    # Check if we are active
            send sign(<"error", request.requestid>) to client       # Sign using replica's private key
            return

        if valid(client.clientKey) == False:                        # Check validity of client key
            send sign(<"error", request.requestid>) to client       # Sign using replica's private key
    return
        orderCommand(operations = request.operation, objectId = request.objectId)


    # Handler for retransmission requests
    handleRetranmissionRequest(client, request):
        if self.status == Immutable:
            send sign(<"error", request.requestid>) to client       # Sign using replica's private key
            return
        else if request.id in cache:                                # Sign using replica's private key
            send sign(<"retranmissionResponse", cache[request.id]>) to client
            return
        else:
            if self.headFlag ==  False:                             # forward the request to head
                sendRequestToReplica(headReplica, request)
                timeout(minutes = self.timeoutSpan):
                    if request.id not in cache:                     # if no result shuttle obtained then reconfig
                        self.status = Immutable
                        send sign(<"reconfigure">) to Olympus       # Sign using replica's private key
                        return
                    Else
                        # Sign using replica's private key
                        send sign(<"retranmissionResponse", cache[request.id]>) to client
                        return
```

```
        else:                                   # we are the head replica
            if request.id in cache:             # if no result shuttle obtained then reconfig
                                                # Sign using replica's private key
                send sign(<"retranmissionResponse", cache[request.id]>) to client
                return
            else if request.id in pendingRequests:
                timeout(minutes = self.timeoutSpan):
                    if request.id not in cache  # if no result shuttle obtained then reconfig
                        self.status = Immutable
                        send sign(<"reconfigure">) to Olympus   # Sign using replica's private key
                        return
                    Else
                        # Sign using replica's private key
                        send sign(<"retranmissionResponse", cache[request.id]>) to client
                        return
            else:                   # similar to new request therefore calling handleRequest
                handleNewRequest(client, request)


# Handler for wedge requests
handleWedgeRequest(request):
    if (decrypt(request) == False)          # using Olympus public key
        return False                        # Do not respond

    # Make sure our status is Immutable
    self.status = Immutable

    # Sign the data and send it to Olympus
    send <"wedgeResponse", self.runningState, self.orderProofs, self.checkPointProof>
                                            To  Olympus


# Handler for a result shuttle
handleResultShuttle(result, resultProof):
    validFlag = True
    for proof in resultProof:
        if valid(proof) != True
                validFlag = False
            break
    if validFlag == False:              # if the return Shuttle is not valid then ignore
                return
    else:
        cache[result.id] = result       # if it is valid then cache the result
```

```
            pendingRequests.delete(result.id)  # delete from the list of pending requests


        # Handler to forward the checkpoint shuttle to the next replica
        handleCheckpointShuttle(CheckPointProof):
            # See proposeCheckpoint for chechpointTuple description
            checkPointTuple = <self.checkpointNum, self.checkPointHash>
            CheckPointProof.append(checkpointTuple)
            if self.tailFlag:
                # Return to the previous replica
                handleCheckpointReturnShuttle(CheckPointProof)
            else:
                # Forward to the next replica
                send <"checkpoint", CheckPointProof> to self.nextReplica



        # Handler to validate the return checkpoint shuttle
        handleCheckpointReturnShuttle(CheckPointProof):
            # Validation
            for checkpointTuple in CheckPointProof:
                if (checkpointTuple.checkpointNum != self.checkpointNum) or
(checkpointTuple.checkPointHash != self.checkPointHash):
                    return Error                              # Do nothing

            # Delete checkpointed proofs from history
            checkpointSlot = CheckPointProof.checkpointTuple.checkpointNum
            delete history[:checkpointSlot]                   # delete history till checkpoint

            # store the returned checkPointProof from the tail
            self.checkPointProof = CheckPointProof

            # Return to the previous replica if you are not the head
            if not self.headFlag:
                send <CheckPointProof> to self.previousReplica


        # Handler for catch up messages from Olympus
        handleCatchupRequest(request):
            # Validate that this request came from Olympus. Decrypt using Olympus' public key
            if not decrypt(request)
                return False
            for order in request.catchupDifference:
```

```
            # Order command and do not forward to the next replica
            orderCommand(order, new OrderProof, new ResultProof, sendFlag = False)

        stateHash = SHA-256(self.runningState)
        send sign(<"catchupResponse", stateHash>) to Olympus      # Sign using replica's private
key


    # Handler for running state requests from Olympus
    handleRunningStateRequest(request):
        # Validate that this request came from Olympus. Decrypt using Olympus' public key
        if not decrypt(request)
            return False
        send sign(<"runningStateResponse", self.runningState>) to Olympus


    # Gets all relevant proofs from the history
    # start from current checkpoint and reach the required slot, using operations in history
    getProofsFromHistory(required_slot):
        proof_statements = self.history[sel.checkpointNum:required_slot]
        return proof_statements


    # Relates to the orderCommand transition at the replica.
    # Assumes we are active -> This is checked in the new request handler
    orderCommand(order, orderProof, resultProof, sendFlag = True):
        # Validate history
        if self.history has slot: # A slot for this command already exists. Cannot issue it twice
            return False

        # Generate a new slot if we are the head
        if self.headFlag == True:
            sNum = self.maxSlotNum+1
        else
            sNum = order.slotNumber

        # Prevent holes in slot numbers
        if self.history does not contain sNum-1:checkpointNum: # We found a hole
            return False

        # Update maxSlotNum
        self.maxSlotNum = sNum
```

```
        # Apply running state
        result = runningState[request.objectId].apply(request.operation)

        # Create proofs
        order = new OrderProof(slotNumber = sNum, operation = request.operation, proofs =
getProofsFromHistory(sNum))
        signedOrder = sign(key = self.privateKey, statement = order)
        orderProof.append(signedOrder)

        signedResult = sign(key = self.privateKey, statement = result)
        resultProof.append(signedResult)

        # Initiate checkpoints after every X orders where X = self.checkpointInterval
        if maxSlotNum is a multiple of self.checkpointInterval:
            checkpoint(maxSlotNum)
            if self.headFlag                    # If we are the head replica, initiate the checkpoint shuttle
              handleCheckpointShuttle(new CheckPointProof())

        if not replica.tailFlag:
           send <"order", order, orderProof, resultProof> to self.nextReplica
        Else:
           # send result to client
           send <"result", requestID, result, resultProof> to request.client
           # send the return shuttle
           send <"ReturnShuttle", result, resultProof> to self.previousReplica
}
```