CSE 535 ASYNCHRONOUS SYSTEMS

Phase 1: Pseudo-code

Submitted By

Prakhar Avasthi
111432497

Rajat Jain
111452437

Entities

1. **Operation** <client_request_id, command>
2. **CurrentAction** <slot_id, Operation>
3. **Timer** <time_id, threshold>
4. **Configuration** <config_id, List<Replica>>
5. **Olympus** <Configuration, List<PublicKey>>
6. **Client** <Olympus, Configuration, Operation, Timer, List<PublicKey>>
7. **Replica** < List<Replica>, List<OrderProof>, List<ResultProof>, List<CheckpointProof>, runningState, History, Timer, Shuttle, replica_id, state, PrivateKey, List<PublicKey>, <lastCheckpoint, state>>
8. **Head extends Replica** < checkPointLimit>
9. **Tail extends Replica**
10. **OrderProof** <"order", Action, Replica>
11. **ResultProof** <"result", Action, hash(result), Replica>
12. **CheckpointProof** <"checkpoint", hash(state)>
13. **History** <action, List<OrderProof>, List<ResultProof>, result>
14. **Shuttle** <action , List<OrderProof>, List<ResultProof>, result>
15. **ForwardShuttle extends Shuttle**
16. **ResultShuttle extends Shuttle**
17. **CheckpointShuttle extends Shuttle**
18. **CheckpointAckShuttle extends Shuttle**
19. **Message** <replica_id, message, <object>>

Pseudo code

Olympus

- **Receive request from client for current configuration**

```
# Provide a configuration to client, if there is no configuration
# then first create a configuration and then return

Olympus → getCurrentConfiguration:
        if(config = = null):
                config ← makeNewConfig(0)
        return config
```

- **Receive request from replica**

```
# If Oympus received "reconfiguration-request"(sent by replica after its timeout),
# then send "Wedge" message to each replica
# and wait for "Wedged" reply from Quoram of replicas

if request is reconfiguration-request:
        for replica in currentConfiguration:
                sendWedgeRequest()
                awaitForQuoram(Wedged)

# if Olympus received "running-state"(replied by replica for get-running-state request)
# then get the crytohash to state received in this message
# compare it with cryptohash sent by this state in "caughtup" message
# if it matches, create a new configuration with history of this replica
# else send "get-running-state to random replica in Quoram"

if request is running-state:
        if cryptoHash == message.object:
                config ← makeNewConfig(Wedged.Quoram.get(replica_id).history)
        else:
                randomReplica ← random(List<Replica>)
                cryptoHash ← CaughtUpQuoram.get(randomReplica.replica_id).object
                sendGetRunningState(randomReplica)
```

- **Received request from Quoram**

```
# If Olympus received Wedged request from Quoram,
# then first create a longesthistory from checkpoint & histories &create its cryptophash
# send each replica in Quoram "catch-up" message with difference of LH & their history
# then wait to receive "caught-up" message from Quoran of replica

if request is Wedged:
        longestHistory ← createLongestHistory(List<PublicKey>)
        cryptoHash ← hash(longestHistory)
        for message in Wedged.Quoram:
                sendCatchUp(longestHistory – replica.history)
                awaitForQuoram(CaughtUp)
```

```
# If Olympus received "caught-up" request from Quoram,
# then check if cryptohash of each replica matches with each other
# if matches, then make a new configuration with LH
# else, select random replica and send "get-running-state" message

if request is CaughtUp:
        for all message1 and message2 in CaughtUp.Quoram if message1.object is equal
        message2.object):
                config ← makeNewConfig(longestHistory)

        else:
                randomReplica ← random(List<Replica>)
                cryptoHash ← CaughtUp.Quoram.get(randomReplica.replica_id).object
                sendGetRunningState(randomReplica)
```

Client

- **Request for new configuration from Olympus**

```
# at start, client will request a new configuration from olympus and assign it to the config

config ← Olympus.getCurrentConfiguration()
```

- **Send the operation(opr) to Head replica of configuration and start a timer**

```
# client will send the operation to the head node and start its timer so that if it will not hear any
# response within threshhold period, it will ask all the replicas for the operation.

sendOperation(opr, Head)
startTimer(timer_id, threshold)
```

- **If timeout, send the operation(opr) to All replicas of configuration**

```
# if client doesnot hear any response in a particular threshhold period, its timer will expire and it
# will send the operation request to every replica in the configuration

for replica in config.replicas:
        sendOperation(opr, replica)
```

- **Receive result from replica**

```
# when client will hear a result back from the replica, it will validate the result and and compares
# their public keys and accept the result if the quoram of repicas send the same result and their
# public keys matching.

validResult ← compare(List<ResultProof>, List<Replica>, List<PublicKey>)
if validResult: acceptResult()
```

- **Poll for new configuration from Olympus**

```
# if timeout occurs on the client end , for liveliness, , it keeps on checking the olympus if new
# configuration is available. If it is available, it will take that configuration and start the
# procedure #again.

if newConfigAvailable:
        config ← Olympus.getCurrentConfiguration()
```

Replica

- **Receive operation from client or replica**

```
if replica is ACTIVE and operation in history:
        sendToClient(List<ResultProof>, result)


# on receiving an operation request, if the replica is in active state but the result is not present in its
#cache

# if replica is not head, then it will forward the operation request to the head and then start its `timer to
# keep track of the time in which it should get an acknowledgement to kill its timer.

# if replica is a head node,  and the operation matches with the current action it was performing, then it
# will start a timer to keep track of  the time in which it should get an operation result response form the
#result shuttle.

# if replica is a head node and the operation doesnot match with its current cction, it will do the following
#steps,

# It will assign an increasing slot number to the operation.
# Then it will create a forward shuttle.
# It will perform the operation.
# It will create the orderproof and result proof and encrypt it.
# It will append the resultproof and resultproof obtained in step 4 in the  forward shuttle's orderproof and
# resultproof array.

# if replica is immutable, it will return an error to the client


else if replica is ACTIVE and operation not in history:
        if replica is not HEAD:
                startTimer(timer_id_replica, replica_threshold)
                forwardRequestToHead()
```

```
else if replica is Head:
    if operation in currentAction:
        startTimer(timer_id_head, head_threshold)

    else if operation not in currentAction:
        currentAction ← assignSlotNumber(slotNo, operation)
        forwardShuttle ← createForwardShuttle(List<OrderProof>, List<ResultProof>,
        action)
        result ← perform(action)
        orderProof ← createOrderProof("order", action, this)
        encryptedOrderProof ← encrypt(orderProof, PrivateKey)
        forwardShuttle.append(List<OrderProof>, encryptedOrderProof)

        resultProof ← createResultProof("result", action, hash(result), replica)
        encryptedResultProof ← encrypt(resultProof, PrivateKey)
        forwardShuttle.append(List<ResultProof>, encryptedResultProof)

        moveShuttle(forwardShuttle, nextReplica)

else if replica is IMMUTABLE:
    sendToClient(null, "error")
```

- **Receive Shuttle**

```
# After receiving a forward shuttle, replica will check order proof for validity
# if valid, then perform operation, add own order proof and result proof in shuttle then
# if replica is Tail, it will send result and proof to client, create result shuttle & move it
# if replica is not tail, it will move shuttle to next replica
# if proof  not valid in shuttle, send checkpoint & history to Olympus in wedge message
# after sending wedge message, it will become immutable

if shuttle instanceof ForwardShuttle:
    for replica in previousReplicas:
        validOrderProof ← compare(List<OrderProof>, List<Replica>, List<PublicKey>)

    if validOrderProof is TRUE:
        result ← perform(action)
        orderProof ← createOrderProof("order", action, this)
        encryptedOrderProof ← encrypt(orderProof, PrivateKey)
        forwardShuttle.append(List<OrderProof>, encryptedOrderProof)

        resultProof ← createResultProof("result", action, hash(result), replica)
        encryptedResultProof ← encrypt(resutProof, PrivateKey)
        forwardShuttle.append(List<ResultProof>, encryptedResultProof)
```

```
                if replica is TAIL:
                        sendToClient(List<ResultProof>, result)
                        resultShuttle ← createResultShuttle(List<OrderProof>, List<ResultProof>,
                        action, result)
                        history.add() ← (action, List<OrderProof>, List<OrderProof>, result)
                        moveShuttle(resultShuttle , previousReplica)

                else if replica is not TAIL:
                        moveShuttle(forwardShuttle, nextReplica)

        else if not validOrderProof:
                object ← List<CheckpointProof>, <lastCheckpoint, runningState>
                message ← <replica_id, "wedged, object>
                sendWedgedRequest(message)
                becomeImmutable()

if shuttle instanceof ResultShuttle:

   # After receiving a result shuttle, it will save result and proofs in historyand kill timer
   # Head replica will also check slot number and if checkpoint limit is reached
   # it will create Checkpoint shuttle, move it to next and state timer

        if replica is not TAIL:
                List<OrderProof> ← resultShuttle.List<OrderProof>
                List<ResultProof> ← resultShuttle.List<ResultProof>
                history.add() ← (action, List<OrderProof>, List<ResultProof>, result)
                moveShuttle(resultShuttle , previousReplica)

                if repica is HEAD and if (action.slot_Id – lastCheckpoint) = = checkPointLimit:
                        checkpointShuttle ← createCheckpointShuttle(slot_Id, List<CheckpointProof>)
                        checkpointProof ← createCheckpointProof("checkpoint", hash(runningState))
                        checkpointShuttle.append(List<CheckpointProof>, checkpointProof)
                        moveShuttle(checkpointShuttle, nextReplica)
                        startTimer(timer_id_checkpoint, checkpointThreshold)

        if timer is running:
                killTimer(timer_id)
                sendResultToClient()

# After receiving a checkpoint shuttle, replica will append their hash of running state
# Tail replica will truncate history & create CheckPointAck Shuttle, move it back to head

if shuttle instanceof CheckPointShuttle:
        checkpointProof ← createCheckpointProof("checkpoint", hash(runningState))
        checkpointShuttle.append(List<CheckpointProof>, checkpointProof)
        if replica is not TAIL:
```

moveShuttle(checkpointShuttle, nextReplica)

else if replica is TAIL:

checkpointAckShuttle ← createCheckpointAckShuttle(slot_Id, List<CheckpointProof>)
lastCheckpoint ← slot_Id
saveState(lastCheckpoint, runningState)
List<CheckpointProof> ← checkpointShuttle.List<CheckpointShuttle>
history.trucate(slot_Id)
moveShuttle(checkpointAckShuttle, previousReplica)

 # After receiving a CheckPointAck shuttle, replica will save proof and truncate history
 # and move the shuttle back to head
 # Head replica will kill its checkpoint shuttle timer

if shuttle instanceof **CheckPointAckShuttle**:

lastCheckpoint ← slot_Id
List<CheckpointProof> ← checkpointAckShuttle.List<CheckpointProof>
saveState(lastCheckpoint, runningState)
history.trucate(slot_Id)

if replica is not Head:

moveShuttle(checkpointAckShuttle, previousReplica)

if replica is Head:

killTimer(timer_id)

- **Receive request from Qlympus**

 # After receiving "wedge" message, replica will send their history, checkpoint and
 # proof to olympus and become immutable

if request is Wedge:

sendWedgedRequest(history, List<CheckpointProof>, <lastCheckpoint, state>)
becomeImmutable()

 # After receiving "catch-up" message, replica will perform operation with catchup
 # history and send running state back to olympus as "caught-up" message

if request is CatchUp:

operations ← request.catch_up_history
tempRunningState ← perform(operations)
message ← <replica_id, "caught_up", hash(tempRunningState)>
sendCaughtUpRequest(message)

 # After receiving "get-running-up", replica will send running state to Olympus

if request is getRunningState:

message ← <replica_id, “running_state”, hash(tempRunningState)>
sendRunningStateRequest(message)

- **If timeout**

  # if replica timeout occurs, it will send the reconfiguration request to olympus
  sendReconfigurationRequestToOlympus()