Asynchronous Systems

**Project Phase 1**

**Pseudo Code for HMAC Shuttle protocol**

Submitted by:

Shalaka Sidmul [111367731]

Naveen Gaddam [111344916]

# **C**lient:

**Client Request Structure:**

Request ID

Operation

Signature of the client

**Client Data Members:**

currentConfiguration: current configuration from Olympus

pendingOperations: Operations which are awaiting response

**run():**

begin:

request\_id = 0

operations = get\_operations() # get valid operations to perform

for operation in operations:

# add request in pending operations

Pending\_operations.append[request\_id]

# perform operation on local data dictionary

perform\_operation(operation)

# build a signed request for the operation

request = build\_signed\_request(request\_id, operation)

sendRequest(request, operation, request\_id)

# increment request id for next request to be sent

request\_id += 1

end

**sendRequest(request, operation, request\_id): *# for executing operation ’O’***

begin:

*# get the latest configuration from olympus*

currentConfiguration = getConfigurationFromOlympus()

*# send a request with the operation to the head of the chain*

sendRequest( operation , currentConfiguration.HEAD\_OF\_THE\_CHAIN)

*# wait for the response for the operation sent*

response = waitForResponse()

*# If a response is received before time out*

if notNull(response):

*# check whether the result received is signed in resultProof by a quorum of replicas in currentConfiguration*

if validate(response.result , response.resultProof) is NOT True:

retransmit(operation)

end if

end if

else if: response is received from Olympus for the current request id:

accept\_response\_from\_olympus()

# valid response is not received within time out

else:

retransmit(operation)

end else

end

**retransmit(operation): # in case of invalid response or time out, retransmit the operation to all the replicas in the configuration**

begin

*# get new configuration from Olympus and re-transmit the operation*

currentConfiguration = getConfigurationFromOlympus()

sendRequest( operation , currentConfiguration.ALL\_REPLICAS)

*#* *wait for response*

response = waitForResponse()

*# check if error statement from an immutable replica*

if response is an error statement from an immutable replica:

*#* *call handlErrorStatement method*

handleErrorStatement(response)

end if

else if response is from replica:

*# check until a valid result is received*

*# valid result: result signed by a quorum of replicas*

while validate(response.result , response.resultProof) is NOT True and NOT TIMEOUT:

*#* *wait for response*

response = waitForResponse()

*# check if error statement from an immutable replica*

if response is an error statement from an immutable replica:

*# call handlErrorStatement method*

handleErrorStatement(response)

break # exit while loop

end if

end while

end else-if

end

**handleErrorStatement(operation): *# retransmit operation to all replicas in case of an error statement received from an immutable replica***

begin

*# re-transmit the operation*

retransmit(operation)

end

**periodicallyCheckForNewConfiguration(): *# periodically check for new configuration from Olympus and retransmit pending operations if current configuration is different from that of received configuration***

begin:

while client is active:

newConfiguration = getConfigurationFromOlympus()

if newConfiguration.id is not equal to currentConfiguration.id:

currentConfiguration = newConfiguration

for each operation in pendingOperations:

getResultForOperation(operation)

end for

end if

wait for a specified interval of time before proceeding further

end while

end

# Replica:

mode:

ACTIVE

IMMUTABLE

PENDING

Head Replica: *# head replica maintains the following variables*

lastSlotNumber

runningState

orderStatement:

<***order***, slotNumber, operation>\_rho, where rho is the replica in the configuration ‘C’

orderProof:

Tuple of order statement of all the replicas in the configuration ‘C’:

<slotNumber, operation, replica, configuration, {<***order***, slotNumber, operation>\_rho’ | rho’ belongs to ‘C’ and rho’ is less than or equal to current replica} >, where rho’ are the replicas preceding the current replica in the configuration ‘C’

resultStatement:

<***result*,** operation, cryptographic hash(‘r’)>\_rho, where rho is the replica in the configuration ‘C’ and ‘r’ is the result computed by the ‘rho’

resultProof:

Tuple of result statement of all the replicas in the configuration ‘C’

<operation, result, replica, configuration, {<**result,** operation, cryptographic hash(‘r’)>\_rho’ | rho’ belongs to ‘C’ and rho’ is less than or equal to current replica} >, where rho’ are the replicas preceding the current replica in the configuration ‘C’

shuttle:

Tuple of orderProof and resultProof:

<orderProof, resultProof>

errorStatement:

<***error***>

replicaHistory:

Set of order proofs for operations applied to the object in consideration

wedgedStatement:

<***wedged,*** history, latestCompletedCheckpointProof, runningState >\_rho, where rho is the replica in the configuration ‘C’

checkpointStatement:

<***checkpoint,*** cryptographic hash(runningState)>\_rho, where rho is the replica in the configuration ‘C’

checkpointProof:

set of signed checkpoint statements of each replica in the configuration ‘C’

checkpointShuttle:

checkpoint proof with a hash of running state

latestCompletedCheckpointProof:

checkpointProof which contains checkpoint statements for all replicas in the chain

**processClientRequest(request): *# handling of client request depending on type of replica [head, non-head]***

begin

if replica is the HEAD of the chain and NOT IMMUTABLE:

*# validate the received request: authenticate client using its public key*

validateRequest(request)

*# proceed forward only if request is valid else terminate the operation*

*# if result for the request is present in the replica’s cache, send the result to client*

if result corresponding to request ID present in cache:

sendResultToClient(result, resultProof)

if request is forwarded by a replica

*# send a receipt to the replica that forwarded the request*

sendReceiptToReplica(result, resultProof)

end if

end if

else if result is not present in the cache:

*#* *the operation is processed, but the head is waiting for the result shuttle of that operation*

if orderStatement for request.operation is present in the orderProof:

resultShuttle = waitForResultShuttle()

if resultShuttle is received:

sendResultToClient(result, resultProof)

if request is forwarded by a replica

*# send a receipt to the replica that forwarded the request*

sendReceiptToReplica(result, resultProof)

end if

return # *end process for this request*

end if

*# if result shuttle does not arrive within timeout*

else:

*# request Olympus for reconfiguration*

reconfigurationRequestToOlympus()

return *# end process for this request*

end else

end if

*# HEAD does not recognize the operation*

*# increment slot number for the received operation*

nextSlotNumber = lastSlotNumber + 1

*# create shuttle that is to be passed along the chain*

shuttle = createEmptyShuttle()

processRequestShuttle(shuttle)

end if

else if replica is NOT HEAD and NOT IMMUTABLE:

*# validate the received request*

validateRequest(request)

*# proceed further only if request is valid*

*# if result for the request is present in the replica’s cache, send the result to client*

if result corresponding to request is present in cache:

sendResultToClient(result, resultProof)

end if

else if result is not present in the cache:

*# forward the request to the head and wait for the response*

forwardRequestToHead(request)

responseFromHead = waitForResponse()

if notNull(responseFromHead): # *if head responds before timeout, forward response to client*

sendResultToClient(responseFromHead.result, responseFromHead.resultProof)

end if

else: *# if head does not respond before timeout, request Olympus for reconfiguration*  reconfigurationRequestToOlympus()

return *# end process for this request*

end else

else if replica is IMMUTABLE:

sendErrorStatementToClient()

end else if

end

**processRequestShuttle(shuttle): *# on receiving request shuttle from predecessor replica***

begin

*# On receiving the shuttle from predecessor replica*

if replica is not HEAD:

*# check whether the slot number has more than one operations associated with it*

if validate(shuttle.orderProof) is NOT TRUE:

reconfigurationRequestToOlympus()

return *# end process for this request*

end if

end if

if replica has its running state:

*# apply operation to the running state and get the result*

result = applyOperation(shuttle.orderProof, runningState)

*# sign the result computed, and append it to the result proof*

appendResultToResultProof(shuttle.resultProof, resultStatement)

end if

*# sign the order statement, and append it to the order proof*

appendOrderToOrderProof(shuttle.orderProof , orderStatement)

if replica is not TAIL:

*# forward the shuttle to the successor replica*

sendShuttle(shuttle, successor replica)

end if

else if replica is TAIL:

*# forward the result along with the result proof to the client*

sendResultToClient(result, resultProof)

*# cache the result shuttle*

cacheResultShuttle(shuttle)

*# send the result shuttle back up the chain*

sendResultShuttle(shuttle, predecessor replica)

end else if

**processResultShuttle(resultShuttle): *# on receiving result shuttle from successor replica***

begin

*#validate the received result shuttle*

if validate(resultShuttle) is NOT TRUE:

reconfigurationRequestToOlympus()

return *# end process for this request*

end if

*# cache the result shuttle*

cacheResultShuttle(shuttle)

if replica is NOT HEAD:

*# send the result shuttle back up the chain*

sendResultShuttle(shuttle, predecessor replica)

end if

end

**waitForResponse():  *# in case of receiving response before timeout return the response, else return NULL***

begin:

start timer

if response received:

terminate timer

return response

end if

else:

return NULL

end else

end

**applyOperation(shuttle.orderProof, runningState): *# obtain result by executing operation on the running state***

begin:

*# obtain result for the operation by executing the operation on the running state*

result = runningState.executeOperation(shuttle.orderProof.operation)

*# append order proof for the said operation to replicaHistory*

replicaHistory.append(shuttle.orderProof)

return result

end

**processWedgeRequestFromOlympus(wedgeRequest): *# handle <wedge-request> from Olympus***

begin

*# replica suspends updating its history by becoming immutable*

replica.mode = IMMUTABLE

*# return a signed wedged statement*

returncreateSigned*W*edgedStatement(wedgedStatement, publicKey)

end

**seedHistory(initHistoryMessage, newConfiguration): *# initialize history and running state for replicas in new configuration***

begin

*# replica is seeded with the initHistory and its mode is changed to ACTIVE*

if replica is in newConfiguration and replica.mode is PENDING:

replicaHistory = initHistoryMessage.initHistory

replica.runningState = initHistoryMessage.runningState

replica.seedCheckpointProof = initHistoryMessage.seedCheckpointProof

replica.mode = ACTIVE

end

**processCheckpointShuttle(checkpointShuttle)*: # on receiving checkpoint shuttle***

begin:

*# check whether checkpoint shuttle is from predecessor replica*

if checkpointShuttle is from a predecessor replica:

*# get the cryptographic hash of running state*

S(runningState) = cryptographic\_hash(runningState)

*# build signed checkpointStatement and append it to the checkpointProof*

checkpointStatement = createCheckpointStatement(S(h))

appendCheckpointToCheckpointProof(checkpointStatement, checkpointShuttle.checkpointProof)

*# in case of TAIL replica, truncate history corresponding to the checkpoint and send shuttle up the chain*

if replica is TAIL:

truncateReplicaHistory(checkpoint)

sendToPredecessor(checkpointshuttle, predecessorReplica)

end if

else: *# for NON-TAIL replicas, send the shuttle to successor*

sendToSuccessor( checkpointshuttle, successorReplica)

end else

end if

*# if checkpoint shuttle is received from a successor, it is contains the completed checkpointProof*

else if checkpointShuttle is from a successor replica:

*# truncate history corresponding to the checkpoint*

truncateReplicaHistory(checkpointShuttle.checkpointProof)

*# cache the checkpointProof*

latestCompletedCheckpointProof = checkpointShuttle.checkpointProof

*# forward shuttle up the chain*

sendToPredecessor(checkpointshuttle, predecessorReplica)

end else

end

initiateCheckpoint(): *# periodically HEAD replica sends a checkpointShuttle down the chain*

begin:

while replica is active

*# when n operations are performed after current checkpoint [ n is a specified constant ]*

if length of suffix of replicaHistory is greater than *n*:

S(runningState) = cryptographic\_hash(runningState)

checkpointStatement = createCheckpointStatement(S(h))

checkpointShuttle = createCheckpointShuttle(checkpointStatement)

latestCheckpointProof = checkpointShuttle.checkpointProof

sendToSuccessor( checkpointshuttle, successorReplica)

end if

wait for some specified time interval

end while

end

**processCatchUpMessageFromOlympus(catchUpMessage): *# process all the operations received through catch-up message and send the cryptographic hash of the resulting running state as a caught-up message***

begin:

for each operation in catchUpMessage:

runningState.executeOperation(operation)

end for

caughtUpMessage = createCaughtUpMassage(cryptographic\_hash(replica.runningState)

sendCauthUpMessageToOlympus(caughtUpMessage)

end

# Olympus:

currentConfiguration = the current configuration: < unique configuration id, series of **2t + 1** ACTIVE replicas >

quorumOfCurrentConfiguration = set of size **t + 1** of replicas from currentConfigration [ where t = number of failures that the system can tolerate ]

replicaKeys = [list of inter-replica signing keys]

**run(): *#setup the system***

replicas = build\_replicas(config)

start\_replicas(replicas)

generate\_replicas\_keys()

send\_init\_to\_replicas(replicas)

**processReconfigurationRequest(<*reconfiguration-request*>): *# on receiving reconfiguration request from a replica***

# validate the received request

if validate(reconfiguration request) is NOT TRUE:

return

end if

wedgedMessages = [list of wedged statements received from replicas in configuration ‘C’]

seedCheckpointProof = seed checkpoint for replicas of new configuration

begin:

*# send wedge request to all replicas in current configuration*

for each replica in currentConfiguration:

sendWedgeRequestToReplica(wedgeRequest, replica)

end for

*# wait for responses from replicas in quorumOfCurrentConfiguration*

wedgedMessages = collectWedgedMessagesFromQuorumOfReplicas()

*# build the initHistory by selecting the longest orderProof for each slot number*

longestHistory = obtainLongestHistory(wedgedMessages)

*# send catch up message to each replica in quorum check whether the replica for which we got the longestHistory is not faulty*

runningStateHash = sendCatchUpMessage(quorumOfCurrentConfiguration, longestHistory)

*# get running state of a randomly selected replica present in the quorumOfCurrentConfiguration*

runningStateOfReplica = getRunningState(replica)

*# calculate cryptographicHash of runningStateOfReplica*

hashOfRunningStateOfQuorumReplica = calculateCryptographicHash(runningStateOfReplica)

*# loop until hashOfRunningStateOfReplicaIInQuorum is equal to previously obtained hash of running state*

while hashOfRunningStateOfQuorumReplica is NOT EQUAL to runningStateHash:

replica = chooseAnotherReplica(quorumOfCurrentConfiguration)

runningStateOfReplica = getRunningState(replica)

hashOfRunningStateOfReplicaInQuorum = calculateCryptographicHash(runningStateOfReplica)

end while

*# create initial history for replicas in new configuration*

initHistoryMessage.initHistory = longestHistory

initHistoryMessage.runningState = runningStateOfReplica

initHistoryMessage.checkpointProof = seedCheckpointProof

sendSeedHistoryToAllReplicasInNewConfiguration(initHistoryMessage, getNewConfiguration())

end

**obtainLongestHistoryAndFindSeedCheckpointProof(wedgedMessages): *# get the longest order proofs for all slot numbers and find the consistent check point proof for seeding with the initial history***

quorumReplicaHistories = set of histories from quorum of replicas

slotNumbers = set of unique slotNumbers present in history of HEAD replica

checkpointProofs = set of checkpointProofs of all replicas in quorum from wedgedStatements

begin

quorumReplicaHistories = wedgedMessages.getHistories()

slotNumbers = getSlotNumbers(quorumReplicaHistories)

*# for each slotNumber get the consistent maximal size orderProof from histories*

for each slotNumber in slotNumbers:

*# check whether orderProof for slotNumber is consistent across all histories in quorumReplicaHistories*

if orderProofConsistent(slotNumber) is FALSE:

*# choose another quorum of wedged messages and repeat the process*

wedgedMessages = getAnotherQuorumWedgedMessages()

return obtainLongestHistory(wedgedMessages)

end if

else:

*# get the maximal size orderProof for slotNumber from quorumReplicaHistories*

maximalOrderProofForSlotNumber = getMaxOrderProof(slotNumber, quorumReplicaHistories)

longestHistory.append(maximalOrderProofForSlotNumber)

*# check whether checkpointProof exists for the slotNumber in consideration AND it is consistent*

*# across all replicas in quorum, if yes, assign it to the seedCheckpointProof*

if checkpointProofs.getCheckpointProof(slotNumber) is NOT NULL and checkpointProofs.isConsistent(slotNumber) is TRUE:

seedCheckpointProof = checkpointProofs.getCheckpointProof(slotNumber)

end if

end else

end for

return longestHistory

end

**NOTE:** The ‘isConsistent(slotNumber)’ API on checkpointProofs checks whether all replicas have issued checkpoint statements for slotNumber greater than or equal to the slotNumber in the argument, assuming the checkpoint proofs for higher slot number have definitely succeeded those with lower slot numbers.

**sendCatchUpMessage(quorumOfCurrentConfiguration, longestHistory): *# forward the catch\_up message to all the replicas in quorum***

begin

*# send the catch up message to all the replicas in quorumOfCurrentConfiguration to get cryptographic hash of the running state*

for each replica in the quorumOfCurrentConfiguration:

sendCatchUpMessageToReplica(longestHistory – wedgedMessages.getHistory(replica))

end for

*# collect the cryptographic hash of the running state from all the replicas in quorum*

runningStateHashes = collectRunningStateCryptographicHashFromAllQuorumReplicas()

*# check whether all replicas have same running state after catching up, if not, select a different quorum and resend catch-up messages*

if all the cryptographic hashes in runningStateHashes are not equal:

quorumOfCurrentConfiguration = chooseAnotherQuorum()

runningStateHashes = sendCatchUpMessage(quorumOfCurrentConfiguration, longestHistory)

end if

return first element of runningStateHashes

end

**processReconfigurationRequest(<*reconfiguration-request*>, proof\_of\_misbehaviour): *# on receiving reconfiguration request from a client***

# validate the received request

if validate(reconfiguration request) is NOT TRUE:

return

end if

else:

processReconfigurationRequest(<reconfiguration-request>)