*The system contains four entities: the Olympus, a configuration, replicated servers and multiple clients.*

*Olympus distributes keys to all clients and replicas in the current configuration. We assume that all messages sent are signed by the sender’s private key and verified at the receiver with the sender’s public key. If a message fails verification, it is dropped.*

*REPLICA*

*Each replica has a mode, state, history (set of order proofs), a cache that contains result information, the latest checkpoint proof (if any) and a checkpoint index.*

*1. replica p receives operation request from client*

**receive** (operation\_request, from= replica or client) {

**if** (operation\_request.type == new\_request) {

**if** (p is head) {

handleNewRequest(operation\_request, this.p, this.C)

}

} **else if** (operation\_request.type == retransmission\_request) {

**//** operation\_request is a retransmission

**if** ((operation\_request.request\_id, result, result\_proof) **in** p.cached\_result) {

**//** p has already cached the result shuttle

**send** ((result, result\_proof), to= client)

} **else if** (p.mode == IMMUTABLE) {

// p is immutable

**send** (error\_statement, to= client)

} **else** {

**if** (p is head){

**if** (p.cached\_result.request\_id contains operation\_request.request\_id) {

**//** p has already performed the operation, but waiting for result

**//** if timeout, send reconfig request

**await** result\_shuttle

**timeout** t: send (reconfiguration\_request, to= Olympus)

**//** get result, send back to client

**send** ((result, result\_proof), to= client)

} **else** {

**//** p doesn't recognize operation, start as new request

handleNewRequest(operation\_request.o, this.p, this.C)

**//** start timer and wait for response

**//** if timeout, send reconfig request

**await** result\_shuttle

**timeout** t: send (reconfiguration\_request, to= Olympus)

**//** get result, send back to client

**send** ((result, result\_proof),  to= client)

}

} **else** {

**//** p is not head

**send** (operation\_request, to= head replica)

**//** start timer and wait for response

**//** if timeout, send reconfig request

**await** result\_shuttle

**timeout** t: send (reconfiguration\_request, to= Olympus)

**//** get result, send back to client

**send** ((result\_proof, result), to= client)

}

}

}

}

**define** handleNewRequest(operation\_request, p, C) {

**//** find smallest slot number s that hasn't been assigned

max\_slot = 0

**for** order\_proof <s', o', \_, \_, \_> **in** p.history {

**if** (s' >= max\_slot) {

max\_slot = s'

}

}

s = max\_slot + 1

**//** create shuttle

o = operation\_request.operation

shuttle\_order\_statement\_list = [ ]

shuttle.order\_proof = <s, o, p, C, shuttle\_order\_statement\_list>

shuttle\_result\_statement\_list = [ ]

shuttle.result\_proof =

<o, operation\_request.id, p, C, shuttle\_result\_statement\_list>

shuttle.id = operation\_request.id

**//** apply operation and add proofs to shuttle

(result, shuttle) = applyOperation(s, o, p, shuttle)

**//** send shuttle down the chain if more than 1 replica exists, else send result to client

**send** (shuttle, to= next replica)

}

**define** applyOperation(s, o, p, shuttle) {

**//** apply <s,o> to current replica

**//** validate if p has correct state and history

**//** check if there is any conflicting tuple <s', o’> in p.state which s == s' but o != o'

**for** **all** <s', o'> **in** p.state {

**if** ((s’ == s) and (o' != o)) {

**//** slot has already been assigned to some operation

**//** this is a misbehavior, send reconfig request to Olympus and exit function

**send** (reconfiguration\_request, to= Olympus)

**return** null

}

}

**//** check if there is any hole

**for** slot\_num = 0 to s {

**if** (<slot\_num, \_> not in p.state) {

**//** there is a hole in the slots

**//** this is a misbehavior, send reconfig request to Olympus and exit function

**send** (reconfiguration\_request, to= Olympus)

**return** null

}

}

**//** create new pair and add to state

slot\_operation\_pair = <s, o>

p.state.append(slot\_operation\_pair)

**//** increase checkpoint index by 1

p.checkpoint\_index++

**//** perform checkpointing when checkpoint index reached predefined interval in configuration

**if** (p.checkpoint\_index == checkpoint\_interval) {

**//** generate checkpoint proof

checkpoint\_proof.statement\_list = [ ]

checkpoint\_statement = p.sign(<"checkpoint", hash(p.state)>)

checkpoint\_proof.statement\_list.append(checkpoint\_statement)

checkpoint\_shuttle.proof = checkpoint\_proof

**//** prefix is a predefined value in configuration, its the number of entries in history to remove

checkpoint\_shuttle.checkpoint\_remove\_prefix = prefix

**send** (checkpoint\_shuttle, to= next replica)

}

**for** all previous replicas:

validate result\_proof and order\_proof in shuttle

**//** run operation and get result

result = findAndPerformOperation(o)

**//** perform orderCommand transition

order\_statement = orderCommandTransition(p, C, s, o, shuttle)

**//** cache request id and result, let result proof be null for now

p.cached\_result.append((shuttle.id, result, null))

**//** add to shuttle order proof

shuttle.order\_proof.shuttle\_order\_statement.append(order\_statement)

**//** create result statement

result\_statement = p.sign(<"result", o, hash(result)>)

**//** add to shuttle result proof

shuttle.result\_proof.shuttle\_result\_statement.append(result\_statement)

**return** (result, shuttle)

}

**define** findAndPerofrmOperation(request) {

operation = Parse request for type of operation to be performed

return operation.result

}

**define** orderCommandTransition(p, C, s, o, shuttle) {

**//** precondition checks

**//** check mode of replica, must be ACTIVE

**if** (p.mode != ACTIVE) {

**//** p is not ACTIVE

**//** since none active replica shouldn't receive any operation request or shuttle

**//** ignore operation and exit function

**return** null

}

**//** check if all replica preceding p has signed order statement in shuttle

**if** not (for all p' preceding p, <"order", s', o'>\_p' in shuttle.order\_proof has s' == s and o == o') {

**//** there exist p' preceding p that doesn't sign order statement for <s, o>

**//** this could be caused by some network error, or the shuttle is sent by faulty replica

**//** ignore operation and exit function

**return** null

}

**//** check if there is any conflict order statement in p.history with the current request

**for** order\_proof <s', o', \_, \_, \_> **in** p.history {

**if** (s' == s and o' != o) {

**//** there exists conflicting order proof in p.history

**//** the request cannot be processed, ignore it and exit function

**return** null

}

}

**//** generate order statement for current replica

current\_order\_statement = p.sign(<"order", s, o>)

**//** append all previous current order statement to order proof

order\_proof = <s, o, p, C, order\_statement\_list>

**for** previous\_order\_statement <"order", s', o'>\_p' in shuttle.order\_proof {

**if** (p' is ahead of p and s' == s and o' == o) {

order\_proof.order\_statement\_list.append(previous\_order\_statement)

}

}

order\_proof.order\_statement\_list.append(current\_order\_statement)

**//** add to p.history

p.history.append(order\_proof)

**return** order\_statement

}

*2. replica p receive shuttle*

**receive**(shuttle, from= replica) {

**//** check validity of the order proof

**if** (for all p' preceding p, shuttle.order\_proof contains <"order", s, o>\_p') and (for any <"order", s', o'>\_p' and <"order", s'', o''>\_p'' in shuttle.order\_proof, s == s' and o == o') {

order\_proof is valid

} else {

**//** there exists some error with the order proof in shuttle

**//** 1) not all previous replicas has included their order statement

**//** 2) there exists conflicting order statement in order proof

**//** this could be caused by some network error, or the shuttle is sent by faulty replica

**//** ignore the shuttle and do nothing

**return**

}

**//** check validity of the result proof

**if** (for all p' ahead of p, shuttle.result\_proof contains <"result", o, \_>\_p') and (for any <"result", o', \_>\_p' and <"result", o'', \_>\_p'' in shuttle.result\_proof, o' == o'') {

result\_proof is valid

} **else** {

**//** there exists some error with the result proof in shuttle

**//** 1) not all previous replicas has included their result statement

**//** 2) there exists conflicting result statement in result proof, ie. different operation

**//** this could be caused by some network error, or the shuttle is sent by faulty replica

**//** ignore the shuttle and do nothing

**return**

}

**//** apply operation and add proofs to shuttle

shuttle = applyOperation(s, o, p, shuttle)

**if** (p is tail) {

**//** send result and result proof to client

**send** (result, result\_proof) to client

// create result\_shuttle and send back to previous replica

result\_shuttle.result\_proof = shuttle.result\_proof

result\_shuttle.id = shuttle.id

**send** (result\_shuttle, to= previous replica)

} **else** {

**//** p forwards the shuttle along the chain

**send** (shuttle, to= next replica)

}

}

*3. replica p receives result shuttle*

**receive** (result\_shuttle, from= replica) {

**//** validate result shuttle

**//** we don't validate the result hash here because the client will validate the result

**if** (no result\_statement **in** result\_shuttle.result\_proof is signed by p) and (p.cached\_result **not contains** result\_shuttle.id) {

**//** there exists some issue with the result shuttle

**//** 1) p should have signed a request statement in the result proof

**//** 2) the result should have already been ordered by p, or p should cache the request id

**//** this could be caused by some network error, or the shuttle is sent by faulty replica

**//** ignore the result shuttle and do nothing

**return**

}

**//** cache result proof along with the already cached request id and result

**//** find the tuple that contains the result request id, and add result proof to it

tuple = p.cached\_result.find(result\_shuttle.id, \_, \_)

tuple[2] = result\_shuttle.result\_proof

**if** (p is not head) {

**send** (result\_shuttle, to= previous replica)

}

}

*4. replica p receives wedge request*

**receive** (wedge\_request, from= Olympus) {

**//** becomeImmutable transition

**if** (p.mode == ACTIVE) { **//** precondition check

p.mode == IMMUTABLE

**//** generate wedged statement

wedged\_tuple = <wedged, p.history>

wedged\_statement = p.sign(<"wedged", p.history>)

wedged\_message.wedged\_statement = wedged\_statement

**//** generate checkpoint proof for wedged message

wedged\_message.checkpoint\_proof = p.checkpoint\_proof

**send** (wedged\_message, to= Olympus)

}

}

*5. replica p receives a check point proof*

**receive** (checkpoint\_shuttle, from= replica) {

**if** (checkpoint\_shuttle.checkpoint\_proof **contains** <checkpoint, hash(state)> for all replica) {

**//** checkpoint proof is complete, remove prefix from history

del p.history[0 to checkpoint\_shuttle.checkpoint\_remove\_prefix]

**//** reset checkpoint index for the replica

p.checkpoint\_index = 0

**//** p stores latest checkpoint proof, which will be included with a wedged statement

p.checkpoint\_proof = checkpoint\_shuttle.checkpoint\_proof

**if** (p is not head) {

**send** (checkpoint\_shuttle, to= previous replica)

}

} **else** {

checkpoint\_statement = p.sign(<"checkpoint", hash(p.state)>)

checkpoint\_shuttle.checkpoint\_proof.statement\_list.append(checkpoint\_statement)

**send** (checkpoint\_shuttle, to= next replica)

}

}

*6. replica p receives a catch up message*

**receive** (catch\_up, from= Olympus) {

operations\_to\_run = catch\_up.catch\_up\_sequence

**for** (op **in** operations\_to\_run) {

**//** apply the to catch up operations, since shuttle is not needed set it to null

applyOperation(op.slot, op.operation, this.p, null)

}

caught\_up\_message.state\_hash = hash(p.state)

**send** (caught\_up\_message, to= Olympus)

}

*7. replica p receives a get\_running\_state message*

**receive** (get\_running\_state, from= Olympus) {

running\_state\_message.state = p.state

**send** (running\_state\_message, to= Olympus)

}

*CLIENT*

*Clients are aware of the identity of the head in the chain. Each request is tagged with a request id and type (new or retransmission).*

*1. Send a request*

**define** sendRequest(operation) { **//** an operation to be performed by the server

**//** obtain current configuration from Olympus

chain = getConfig(configQuery, Olympus)

**//** generate operation request message

operation\_request.type = new\_request

**//** client will obtain an id for the request and a specific operation to run

operation\_request.request\_id = new id

operation\_request.operation = new operation

**//**send operation to head of chain

**send** (operation\_request, to= chain.head)

**//** wait for a pair of result and result proof, if timeout, retransmit request

**await** (result, result\_proof)

**timeout** t: retransmit(operation\_request, to = allReplicas)

**//** result\_proof <result, hash(result)> received, verify result

**if** (verify (result, result\_proof)) {

**return** result

}

else{

retransmit(operation)

}

}

**define** retransmit(operation\_request, allReplicas) { **//** retransmit operation\_request

**//** operation request type should now be a retransmission

operation\_request.type = retransmission

**//** send to all replicas

**send** (operation\_request, to= allReplicas)

**//** wait for result from any replica

**await** message from any replica

**if** (message == (result, result\_proof)) {

**//** if an actual result received, verify the result

**for** every result\_proof **in** received{

**if** (verify (result,result\_proof))

**return** result

}

}

} **else if** (message == error\_statement) {

**//** if an error statement received, obtain new config and restart

chain = getConfig(configQuery, Olympus)

retransmit (operation\_request, Olympus.replicas)

}

}

**define** verify(result, result\_proof) { **//** check if result proof is validated

**//** client needs to check that at least t+1 (quorum size) of replicas has matching hash

**if** (hash(result) == result\_statement.hashed\_result of t+1 distinct replicas) {

**return** true

} **else** {

**//** since there doesn't exist t+1 replicas with matching result hash, result is not validated

**//**send reconfiguration request along with proof of misbehavior

reconfiguration\_request.proof\_of\_misbehavior = (result, result\_proof)

**send** (reconfiguration\_request, to= Olympus)

}

}

**define** getConfig(configQuery, Olympus) { **//** client gets current configuration of chain from Olympus

**send** (configQuery, to= Olympus)

**await** response

**//** response should contain a list of replica, with information of the head replica

chain = response.getReplicaList

**return** chain

}

*OLYMPUS*

*Olympus is responsible for coordination. It is a fail tolerant, centralized configuration service that generates a series of configurations and initializes them.*

*1. setup for a configuration*

**define** setup (configuration C) {

**//** generate keys that are used to sign statements

**for** each replica p **in** C {

createProcess()

startProcess()

createKey()

**send** (key, to= p)

}

**for** each client c **in** C {

createProcess()

startProcess()

createKey()

**send** (key, to= c)

}

**//** initialize C

C.setup()

**for** all p **in** C{

becomeActive(p, this.C, [], [])

}

}

*2. receive reconfiguration request from either replicas or clients*

**receive** (reconfiguration\_request, from= sender){

**//** if sender is a client

**if** (sender == client) {

verify proof\_of\_misbehaviour

**if** invalid {

**//** proof of misbehavior is not validated, just ignore the request

**return**

}

}

**//** request wedge statements from replicas in a configuration, replicas perform becomeImmutable()

**send** (<wedge\_request>\_olympus, to= allReplicas)

**//** wait for wedged message from all replica

**await** wedged\_message from allReplica

C’ = nextConfiguration

change\_confi\_successful = false

**while** (!change\_config\_successful) { **//** find a quorum Q of replicas that is consistent

**for** a quorum Q of replicas {

H = { <wedged\_statement>\_p | p belongs to Q }

change\_config\_successful = switchConfig (C, C’, H, wedged\_message)

}

}

}

*2. receive check configuration request from either clients*

**receive** (check\_configuration, from= client){

send(('check\_configuration\_response', 'OK', message\_id, self), to = client)

}

**define** switchConfig(C,C’,H, wedged\_message){

**//** return true if hist is created from a valid quorum of replicas

**//** precondition check

**if** (succ(C, C’) and (C’ has not been initialized) and (|H| = |Q|) and (for p all in Q: <wedged\_statement>\_p in wedged\_message belongs to H) and (Q is consistent) and validHist(<wedged\_statement>\_p.history, p, C)) {

LH = longest history of a replica in Q

**for** all p **in** Q {

**//** validate checkpoint proof in wedged\_message

validate (wedged\_message.checkpoint\_proof)

**//** generate catch up message

**//** wedged\_statement is taken from wedged\_message

catch\_up.catch\_up\_sequence = LH - <wedged\_statement>\_p.history

**send** (catch\_up, to= p)

**//** wait for caught up message from replica

**await** caught\_up\_message

p.ch = caught\_up\_message.state\_hash

}

**//** checks cryptographic hash ch in the caught\_up messages from all replicas in Q

**for** all p **in** Q {

**//** if one of the replicas in Q is faulty, then Q is not the desired quorum

**if** (p.ch != p'.ch for any other replica p' in Q) {

**return** false

}

}

running\_state = null

**//** check ch of any random replica

**//** if ch are equal, Olympus includes hist as the initial running state of the new config

**//** else request from another replica

**while** (hash(running\_state) != p'.ch for any other replica p') {

**//** send get running state message

**send** (get\_running\_state, to= random(p in Q))

**//** wait for running state message

**await** running\_state\_message

running\_state = running\_state\_message.state

}

hist = order proofs of maximal size from quorum Q

**//** generate initHist message

initHist\_message.initHist\_statement = Olympus.sign(<initHist, C’, hist>)

initHist\_message.initial\_state = running\_state

**//** send to new configuration C'

**send** (initHist\_message, to= C')

**return** true

}

**return** false

}

**define** validHist (wedged\_statement, p, C) {

hist = wedged\_statement.history

**//** check if there is any conflict in history

**if** (for any order proof <s, o, p, \_, list> and <s', o', p', \_, list'> where s == s' and (o != o' or p != p' or list != list') {

**//** there is a conflict in history of wedged statement

**return** false

}

**//** check if every order proof has proper order statements

**for** (all order\_proof <s, o, p, \_, \_> **in** hist) {

**if** not (**for** all p' preceding p, <"order", s, o>\_p' exists **in** order\_proof) {

**//** not all order statement from previous replicas exist in order proof

**return** false

}

}

**//** wedged\_statement is validated

**return** true

}

*3. receive request to get configuration from client*

**receive** (configQuery, from= client){

**send** (C, to= client) **//** reply with current configuration C

}

*CONFIGURATION*

*Contains a list of replicas in the current configuration and provides them with the correct history.*

**define** setup() { **//** setup for configuration C

**for** each replica p **in** configuration C {

p.mode = PENDING

p.history = NULL

p.state = NULL

}

}

*1. receive initHist message*

**receive** (initHist\_message, from= Olympus) {

**//** validate initHist message, ie. check if it is signed by Olympus

**if** (validate(initHist\_message)) {

**//** perform becomeActive transition for every replica in C

**for** all p **in** C {

**//** precondition check

**if** ((p is in C) and (p.mode == PENDING) and (<"initHist",C, hist>\_olympus in received(initHist\_statement) of C)) {

becomeActive(p, this.C, initHist\_message.initHist\_statement.hist, initHist\_message.initial\_state)

}

}

}

}

**define** becomeActive (p, C, hist, state) { **//** becomeActive transition for replica p in configuration C

**if** (p.mode == PENDING){

p.history = hist

p.state = state

p.mode = ACTIVE

}

}