Project: Byzantine Chain Replication **PHASE 1: PSEUDO-CODE**

Name: Sharad Gupta (111500752) , Yash Shah (111485295)

Assumptions:

1.) There exists only one olympus server, one client and 2t+1 replicas
2.) Clients sends its request for operations to head usually in failure free cases.
3.) Some of the initialization variables are statically defined in the classes
4.) Some of the functions are not defined assuming they are very basic to write

########## **Client** ##########

Class Client :

__init__main ():

```
self.olympusId = //Statically defined
self.current_config = get_configuration_from_olympus(self.olympusId)
self.request_timer_map = {}
self.clientId = get_clienId()
// get_replicas_public_key() gets all replica's public keys from current config
self.replica_public_keys = get_replicas_public_key(self.current_config)
// This map allocates timer value to each request
self.timer_threshold_map = {}
self.requestId_to_request_map = {}
start_listening_for_application_requests()
```

/* This function is used to listen indefinitely to application requests, start the timer for them, check for their expiration and also send requests to head. It also generates threshold generates the threshold based on the request size */

def start_listening_for_application_requests ():

```
while True:
    check_liveness_of_current_config()
    check_for_all_timers()
    request  = get_next_request_from_application()
    If request is not empty:
        requestId = generate_new_requestId()
        process_request(requestId, request)
```

/* This function is used to process a new request on client side */

def process_request (requestId, request):

```
timer = start_timer()
self.requestId_to_request_map[requestId] = request
self.request_timer_map[requestId] = timer
self.timer_threshold_map[requestId] = generate_threshold()
head = get_head_from_current_config()
send(request = "execute_operation_of_client",
param={requestId, request, clientId}, to = head)
```

```
/* This function gets configuration object from the olympus using its id */
def get_configuration_from_olympus ():
        // config object contains sequence no of config and ordered list of replicas info
        config  = receive(request="get_current_config", to=self.olympusId)
        return config

/* This function receives result of operation from tail  and checks its validity, if found
conflicting then transmit misbehaviour to olympus */
 def recieve_result_for_the_operation (requestId, result_proof, result):
        If requestId in self.request_timer_map:
                check_flag = check_the_validity_of_result_proof(result_proof, result)
                stop_timer(requestId)
                remove_from_request_timer_map(requestId)
                If !check_flag:
                        send(request="recieve_proof_of_mishbehaviour_from_client",
                        params={result_proof, result}, to = self.olympusId)


/* This function runs indefinitely and check for all existing timers if expired or not , if
expired retransmits the operation to all replicas */
def check_for_all_timers ():
        /* Run a while loop and check for all timers , if expired retransmit to all
        replicas and if threshold of retries crossed drop request */
        for each requestId in self.request_timer_map:
                time_spent  = current_time - self.request_timer_map[requestId]
                If time_spent > self.timer_threshold_map[requestId]:
                        retransmit_to_all_replicas(
                                self.requestId_to_request_map[requestId], requestId)
                        remove_from_request_timer_map(requestId)

/* This function retransmit the operation to all replicas in case of failures */
def retransmit_to_all_replicas (request, requestId):
        replicas = get_all_replicas_from_config(self.current_config)
        for each replica in replicas:
                send(request = "execute_retransmitted_operation_of_client",
                param={requestId, request, self.clientId}, to = replica)

/* This function checks for new config periodically and if a head is updated then
retransmit pending operations */
def check_liveness_of_current_config ():
        //call olympus to get new config and check
        current_config = get_configuration_from_olympus(self.olympusId)
        If get_head(current_config) != get_head(self.config):
                for each requestId in self.requestId_to_request_map:
                        retransmit_to_all_replicas(
                                self.requestId_to_request_map[requestid], requestId)
```

```
/* This function checks the validity of result proof at the client side */
def check_the_validity_of_result_proof (result_proof, result):
        resulthash = hash(result)
        for index in range(0,len(result_proof)):
            if check_valid_signature(
                result_proof[index], self.replicas_public_keys[index]) :
                // or_pair is a tuple of operation and hash of result
                if resulthash != result_proof[index].or_pair.second:
                    return false
            else:
                return false
        return true


/* This function receives result for the retried operation and in case of error get latest
config from olympus and again retries */
def recieve_result_for_retried_operation (requestId, result_proof, result, type):
    If type == “ok”:
        check_flag = check_the_validity_of_result_proof(result_proof, result)
        If !check_flag:
            send(request=”recieve_proof_of_mishbehaviour_from_client”,
            params={result_proof, result}, to = self.olympusId)
    elif type == “error”:
        self.config = receive(request=”get_current_config”, to=olympusId)
        process_request(requestId,
        self.requestId_to_request_map[requestId])
```

################################## **Replica** ##################################

Class Replica:

__init__main():

```
// types of replica nodes
enum replicaType(head, internal, tail)
// types of replica modes
enum mode(pending, active, immutable)
self.history_of_order_proof = {}
// before intihist statement from olympus in configuration, mode is pending
self.mode = mode.pending
self.olympusId = //Statically defined
// get current configuration object from olympus
self.config = receive(request="get_current_config", to=olympusId)
self.replicaId = //Statically defined
self.request_timer_map = {}
self.checkpoint_proof = []
// The cache of recently executed request's Id
self.requestId_cache = []
// get public key of olympus from configuration object returned by olympus
self.olympus_public_key = get_olympus_key_from_config(self.config)
// get head's id from configuration object returned by olympus
self.headId = get_head_from_config(self.config)
// the map of result and result_shuttle cache for each requestId
self.result_and_result_shuttle_cache = {}
self.running_state = //running state object
// the map of timer threshold for each request's Id
self.timer_threshold_map = {}

/* get replicaType, successorId and predecessorId for current replica using
configuration object received from olympus */
self.replicaType = get_replica_type(self.config, self.replicaId)
self.succesorId = get_successorId(self.config, self.replicaId)
self.predecessorId = get_predecessorId(self.config, self.replicaId)

/* get_replicas_public_key returns order list of tuple of replica and public key
using the configuration object received from olympus */
self.replicas_public_keys = get_replicas_public_key(self.config)

// The below functions are executed periodically
drop_requestId_cache_periodically()
drop_result_and_result_shuttle_cache_periodically()
initiate_checkpoint_periodically()
check_for_all_timers_periodically()
```

```
/* This function receives operation on head from the client and perform action on it */
def execute_operation_of_client (requestId, operation, clientId):
        shuttle = get_empty_shuttle()
        slotno = get_next_slot_for_operation()
        so_pair = (slotno, operation)
        execute_operation(requestId, operation, clientId, shuttle, so_pair)

/* This function receives retried operation from a replica and perform action on it */
def execute_retried_operation_of_client (requestId, operation, clientId):
        If requestId in self.result_and_result_shuttle_cache:
                result_proof =
                        self.result_and_result_shuttle_cache[requestId].first
                result =self.result_and_result_shuttle_cache[requestId].second
                send_result_to_client(requestId,result_proof, result)
        elif (requestId not in self.result_and_result_shuttle_cache)
                        and (requestId in self.requestId_cache) :
                timer = start_timer()
                self.request_timer_map[requestId] = timer
                self.timer_threshold_map[requestId] = generate_threshold()
        else:
                timer = start_timer()
                self.request_timer_map[requestId] = timer
                self.timer_threshold_map[requestId] = generate_threshold()
                shuttle = get_empty_shuttle()
                slotno = get_next_slot_for_operation()
                so_pair = (slotno, operation)
                execute_operation(
                        requestId, operation, clientId, shuttle, so_pair)

/* This function is the actual implementation of action on operation which calculates
result and also forward the shuttle to next replicas */
def execute_operation (requestId, operation, clientId, shuttle, so_pair):
        If self.mode == mode.active:
                checkflag  = check_the_validity_of_order_proof(shuttle)
                If checkflag:
                        result  = self.running_state.exec(operation)
                        add_order_proof(shuttle, so_pair)
                        add_result_proof(shuttle, operation, result)
                        self.requestId_cache.append(requestId)
                        self.history_of_order_proof[so_pair] =
                        shuttle.order_proof
                        if self.replicaType != replicaType.tail:
                                forward_shuttle_to_next_replica(
                                  requestId, operation, clientId, shuttle, so_pair)
                        else:
                                // Send the result with result proof to client
```

```
                                        send_result_to_client(
                                                requestId,shuttle.result_proof, result)
                                        send_result_shuttle_to_preceding_replica(
                                                shuttle, result, requestId)

                        else:
                                send_reconfigure_request_to_olympus(requestId)
                elif self.mode == mode.immutable:
                        send_reconfigure_request_to_olympus(requestId)

/* This function sends reconfiguration request to olympus and removes requestId
specific timer from map */
def send_reconfigure_request_to_olympus(requestId):
        remove_from_request_timer_map(requestId)
        send(request="recieve_reconfiguration_request_from_replica",
        params={self.replicaId}, to = self.olympusId)


/* This function receives the result shuttle and cache it if found valid else send
reconfigure request to olympus */
def execute_shuttle_recieved_from_succeding_replica (
                shuttle, result, requestId):
        If self.mode == mode.active:
                check_flag = check_the_validity_of_result_proof(shuttle, result)
                If check_flag:
                        cache_shuttle_and_result(shuttle, result, requestId)
                        If self.replicaType != replicaType.head
                                send_result_shuttle_to_preceding_replica(
                                        shuttle, result)
                        If requestId in self.request_timer_map:
                                time_spent  = current_time -
                                self.request_timer_map[requestId]
                                If time_spent < self.timer_threshold_
                                        map[requestId]:
                                        remove_from_request_timer_map(requestId)
                                        send_result_to_client(
                                                requestId, result_proof, result)
                else:
                        send_reconfigure_request_to_olympus(requestId)
        elif self.mode == mode.immutable:
                send_reconfigure_request_to_olympus(requestId)
```

```
/* This function checks the validity of result proof in the result shuttle */
def check_the_validity_of_result_proof(shuttle, result):
        resulthash = hash(result)
        for index in range(0,len(shuttle.result_proof)):
            if check_valid_signature(
            shuttle.result_proof[index], self.replicas_public_keys[index]) :
                If resulthash !=
            shuttle.result_proof[index].second.second:
                    return false
            else:
                return false
        return true


/* This function checks the validity of order proof in the shuttle */
def check_the_validity_of_order_proof(shuttle):
        so_pair = get_first_so_pair_from_order_proof(shuttle.order_proof)
        for index in range(0,len(shuttle.order_proof)):
            if check_valid_signature(
            shuttle.order_proof[index], self.replicas_public_keys[index]) :
                If so_pair != shuttle.order_proof[index].second:
                    return false
            else:
                return false
        return true

/* This function adds order proof to the ongoing shuttle by signing the so_pair using
replica's key */
def add_order_proof(shuttle, so_pair):
    shuttle.order_proof.append(
        sign_tuple_using_key("order",so_pair, self.replicaId))

/* This function encrypts the result using crypt function and adds result proof to
shuttle by signing the or_pair using replica's key */
def add_result_proof(shuttle, operation, result):
    or_pair = (operation, hash(result))
    shuttle.result_proof.append(
        sign_tuple_using_key("result",or_pair, self.replicaId))

/* This function forwards the shuttle to the successor replica in the chain */
def forward_shuttle_to_next_replica(
    requestId, operation, clientId, shuttle, so_pair):
    send(request="execute_operation",
    params={requestId, operation, clientId, shuttle, so_pair}, to=self.successorId)
```

```
/* This function sends result_proof and result to the client */
def send_result_to_client(requestId,result_proof, result):
        send(request="recieve_result_for_the_operation",
        params={requestId,result_proof, result}, to=self.clientId)

/* This function sends result and shuttle to preceding replica in the chain */
def send_result_shuttle_to_preceding_replica(shuttle, result, requestId):
        send(request="execute_shuttle_recieved_from_succeding_replica",
        params={shuttle, result, requestId}, to=self.predecessorId)

/* This function caches the result shuttle and result itself to deal with failures */
def cache_shuttle_and_result(shuttle, result, requestId):
        self.result_and_result_shuttle_cache[requestId] = tuple(shuttle, result)

/* This function is called when replica finds the misbehaviour and becomes
immutable along with sending wedged statements to olympus */
def become_immutable():
        If self.mode == mode.active:
                self.mode = mode.immutable
                send(request="recieve_wedge_response_from_replica",
                params={sign_tuple_using_key(self.history_of_order_proof,
                self.checkpoint_proof), self.replicaId}, to = self.olympusId)

/* This function is used to receive operation from the client in case of failure and send
the result if cached */
def execute_retransmitted_operation_of_client(requestId, request, clientId):
        If requestId in self.result_and_result_shuttle_cache:
                        result_proof =  self.result_and_result_shuttle_
                        cache[requestId].first
                        result =self.result_and_result_shuttle_cache[requestId].second
                        send(request="recieve_result_for_retried_operation",
                        params={requestId,result_proof, result, "ok"}, to = clientId)
        elif self.mode == mode.immutable:
                 send(request="recieve_result_for_retried_operation",
                params={requestId,null, null,"error"}, to = clientId)
        else:
                If self.replicaType != replicaType.head:
                        timer = start_timer()
                        self.request_timer_map[requestId] = timer
                        send(request="execute_retried_operation_of_client",
                        params={requestId, operation, clientId}, to = headId)

/* This function is used to return the current running state of replica to the olympus */
def get_running_state():
        send(request="recieve_running_state", params={hash(self.running_state),
        replicaId}, to = self.olympusId)
```

```
/* This function runs indefinitely and check for all existing timers if expired or not , if
expired send reconfiguration request to olympus */
def check_for_all_timers_periodically():
        /* run a while loop and check for all timers , if expired retransmit to all replicas
        and if threshold of retries crossed drop request */
        for each requestId in self.request_timer_map:
                time_spent  = current_time - self.request_timer_map[requestId]
                If time_spent > self.timer_threshold_map[requestId]:
                        send_reconfigure_request_to_olympus(requestId)

/* This function receives wedge request from olympus and checks for valid signature,
if valid calls become immutable */
def recieve_wedge_request_from_olympus(wedge_statement):
        if check_valid_signature(wedge_statement,self.olympus_public_key):
                        become_immutable()

/* This function is used to make the replica active and update its running state */
def become_active(running_state, history):
        if check_valid_signature(history, self.olympus_public_key):
                self.mode = mode.active
                self.running_state = running_state
                self.history = history

/* This function is used to receive checkpoint shuttle from the preceding replica ,
forward to next replica, also if tail encountered calls delete history and return the
shuttle in reverse direction */
def recieve_checkpoint_shuttle_from_preceding_replica(checkpoint_shuttle):
        running_state_hash = hash(self.running_state)
        slot_removal_counts = checkpoint_shuttle.checkpoint_proof.third_element
        checkpoint_shuttle.checkpoint_proof.append(
                sign_tuple_using_key(
                        “checkpoint”,running_state_hash, slot_removal_counts))
        If self.replicaType != replicaType.tail:
                send(request=”recieve_checkpoint_shuttle_from_preceding_replica”,
                params={checkpoint_shuttle}, to=self.successorId)
        else:
                delete_history_using_checkpoint(
                        checkpoint_shuttle.checkpoint_proof)
                self.checkpoint_proof = checkpoint_shuttle.ckeckpoint_proof
                send(request=”recieve_checkpoint_shuttle_from_succeeding_replica”,
                params={checkpoint_shuttle}, to=self.predecessorId)
```

```
/* This function is used to receive checkpoint shuttle from the succeeding replica and
process it */
def recieve_checkpoint_shuttle_from_succeeding_replica(checkpoint_shuttle):
        If self.mode == mode.active:
                delete_history_using_checkpoint(
                        checkpoint_shuttle.checkpoint_proof)
                self.checkpoint_proof = checkpoint_shuttle.checkpoint_proof
        elif self.mode == mode.immutable:
                        send(request="recieve_reconfiguration_request_from_replica",
                        params={self.replicaId}, to = self.olympusId)


/* This function is used to initiate the checkpoint periodically from the head */
def initiate_checkpoint_periodically():
        If self.replicaType = replicaType.head:
                checkpoint_shuttle = create_empty_checkpoint_shuttle()
                checkpoint_shuttle.checkpoint_proof.append(
                        sign_tuple_using_key("checkpoint", hash(self.running_state),
                        self.slot_removal_counts)
                send(request="recieve_checkpoint_shuttle_from_preceding_replica",
                params={checkpoint_shuttle}, to=self.successorId)

/* This function is used to delete history using checkpoint proof received from the
chain */
def delete_history_using_checkpoint(checkpoint_proof):
        /* Here third element is the slot_count i.e amount of slot we have to truncate
        from the history */
        slot_removal_counts = checkpoint_proof.third_element
        oldest_slot_no = get_oldest_slot_no(self.history_of_order_proof)
        for each so_pair in self.history_of_order_proof:
                /* so_pair is a tuple of slot number and operation, so_pair.first access
                slot number */
                If so_pair.first_element is in the range of (
                        oldest_slot_no, oldest_slot_no + slot_removal_counts):
                        remove so_pair from the self.history_of_order_proof

/* This function is used to receive catch up message from olympus and send  the
running state after executing operations to olympus */
def catch_up(operations):
        self.running_state.exec(operations)
        send(request="caught_up", params={hash(self.running_state), replicaId},
        to = self.olympusId)
```

################################## **Olympus** ##################################

Class Olympus:

___init_main():

self.replicasId = []
self.replica_public_keys = {}
self.replicaId_to_public_key_map = {}
self.olympus_private_key = //statically defined
self.wedge_responses_from_replicas = {}
self.quorum_size = //Statically defined
self.configuration_no = //Statically defined
self.state_hashes=[]
self.running_state = null
self.configuration_in_process_flag = false

/* This function creates new configuration of replicas and running state, for first time the state will be empty but for consecutive it will pass on the latest checkpointed state */

def generate_new_configuration(running_state , history):

self.replicasId = generate_new_replicas()
self.configuration_no = get_new_config_no()
for each replicaId in self.replicasId:
send(request="become_active",
params={sign_using_key(running_state, history)}, to = replicaId)

/* This function spawns new processes on the current replicas server , generate new public, private keys of replicas and also sends them these keys */

def generate_new_replicas()

spawn_processes_on_replicas()
keys_object = generate_public_private_keys_for_replicas()
self.replica_public_keys = get_public_keys(keys_object)
self.replicaId_to_public_key_map =
create_replicaId_to_public_key_map(self.replicasId, self.replica_public_keys)

/* This function returns the current configuration no and order list of replicas Info of olympus */

def get_current_config():

/* There exists a create_replicas_info which contains all sort of replicas information in it such as id, public key of all , private key */
return (self.configuration_no, create_replicas_info())

/* This function checks for validity of result proof of client */

def recieve_proof_of_mishbehaviour_from_client(result_proof, result):

check_flag = check_the_validity_of_result_proof(result_proof, result)
If !check_flag:
//Do nothing because we have assumed there is no byzantine client

```
/* This function checks the validity of result proof at the olympus */
def check_the_validity_of_result_proof(result_proof, result):
        resulthash = hash(result)
        for index in range(0,len(result_proof)):
            if check_valid_signature(
                result_proof[index], self.replicas_public_keys[index]) :
                if resulthash !=result_proof[index].or_pair.second:
                    return false
            else:
                return false
        return true

/* This function is used for listening reconfiguration request from replicas */
def recieve_reconfiguration_request_from_replica() :
        If self.configuration_in_process_flag == false:
            send_signed_wedge_request_to_all_replica()
            self.configuration_in_process_flag = true

/* This  function is  used to send signs wedge statement with key to all replicas in
configuration */
def send_signed_wedge_request_to_all_replica():
    signed_wedge = sign_using_key("wedge")
    for each replicaId in self.replicasId:
        send(request="recieve_wedge_request_from_olympus",
        params={signed_wedge}, to = self.replicaId)

/* This function receives wedge response from replicas after checking valid signature
and then tries to find valid quorum based on responses */
def recieve_wedge_response_from_replica(wedged_history, replicaId):
    If check_valid_signature(
        wedged_history, self.replicaId_to_public_key_map[replicaId]):
        // wedged_history is a tuple of replica history and checkpoint proof.
        self.wedge_responses_from_replicas[replicaId] = wedged_history
        check_if_any_valid_quorum_exists()


/* This function checks validity of quorum and generates a valid quorum from wedge
responses from replica in order to proceed with reconfiguration */
def check_if_any_valid_quorum_exists():
    If len(self.wedge_responses_from_replicas) >= self.quorum_size:
        /* generate_quorums_from_current_wedge_responses function
        creates all possible quorums from the current replicas responses */
        list_of_quorums =
        generate_quorums_from_current_wedge_responses()
```

```
while list_of_quorums not empty:
        quorum = list_of_quorums.get_first()
                If check_valid_history_in_current_quorum(quorum) and
                check_valid_checkpoint_proof_in_current_quorum(
                        quorum):
                /* This function gets the longest running state replica
                from the given quorum */
                longest_running_state_replica=
                get_longest_running_state_replica_from_quorum(
                        quorum)

                catch_up(quorum, longest_running_state_replica)
                await caught_up from all replicaId in quorum

                for each replicaId in quorum:
                        If replica running state hash !=
                        hash(longest_running_state)
                        continue
                        // Try next quorum as there is a mismatch in
                        //the hash of one of the replica

                for each replicaId in quorum:
                        hash_of_state = get_running_state()
                        If hash_of_state ==
                        hash(longest_running_state):
                                break
                        // found some replica with longest running state
                        break the loop

                // This kills all old replicas processes
                kill_all_old_replicas()
                self.running_state = running_state
                /*history is set to null as the running state is updated to
                catch up to that */
                generate_new_configuration(running_state, null)
                self.configuration_in_process_flag = false
list_of_quorum.remove(quorum)
```

```
/* This function is used to receive caught_up message from the replica during
reconfiguration */
def caught_up(hash_of_state, replicaId):
        self.state_hashes.append(tuple(replicaId, hash_of_state))
```

```
/* This function is used to send catch_up message from the replica */
def catch_up(quorum, longest_running_state_replica):
        for each replicaId in quorum:
                history_diff = get_history_diff_between_two_replicas(
                        longest_running_state_replica, quorum[replicaId])
                send(request="catch_up", params={history_diff}, to = replicaId)

/* This function is used to check the validity of checkpoint proof of replicas in quorum
*/
def check_valid_checkpoint_proof_in_current_quorum(quorum):
        If checkpoint_proof is equal for all replicas in quorum:
                return true
        else:
                return false

/* This function is used to check the validity of histories of replicas in quorum */
def check_valid_history_in_current_quorum(quorum):
        /* Here it checks for the validity of order proof in every replicas i.e if there
        exist at most one order proof for every slot in the quorum */
        for each replicaId in quorum:
                history = self.wedge_responses_from_replicas[replicaId].first
                If not check_for_at_most_one_order_proof_per_slot(history):
                        return false
        return true

/* This function is used to construct the history by selecting longest order proof for
each slot in quorum */
def create_history_from_quorum(quorum):
        new_history = {}
        for each replicaId in quorum:
                history = self.wedge_responses_from_replicas[replicaId]
                for each so_pair in history :
                        order_proof = history[so_pair]
                        If len(order_proof) > len(new_history[so_pair])
                                new_history[so_pair] = order_proof
        return new_history
```