CSE 535: Asynchronous Systems

Phase – 1 Submission

Team members:

Neel Paratkar	111483570	nparatkar@cs.stonybrook.edu
Jay Torasakar	111406252	jtorasakar@cs.stonybrook.edu

CLIENT

#Has a unique identifier CLID : client identifier

Client for executing an operation 'O' does:

```
send("getConfig", CLID, Olympus)                    #sends getConfig message to Olympus
receive("current_config", current_config)           #receives current_config message
                                                    alongwith current configuration

executeOperation(O, current_config, current_config.head)
```

FUNCTION**: executeOperation**(O, current_config, replica)

```
# Client sends a Unique Identifier for an operation to the replica
UID = generateUID()
res = send("execute", O, UID, CLID, replica)        #sends execute message to 'replica'
start_timer()
while timer.time < TIMEOUT && res == NULL :
    if timer.time % (TIMEOUT/5) == 0 :              #current time is multiple of TIMEOUT/5
        temp_config = GetLatestConfigFromOlympus()  #polling to check if there is a new
                                                    configuration
        if current_config == temp_config:
            continue()
        else                                        #if a new configuration is available
            executeOperation(O, temp_config, temp_config.head)
                                                    #retransmit the operation

if res == NULL                                      #Didnot receive result from tail in time
    broadcast_operation(O, current_config)
```

FUNCTION: **broadcast_operation**(O, current_config):

```
for every replica in current_config:
    executeOperation(O, current_config, replica)
```

REPLICA

1.On receiving “execute” message:

executeOperation(O, UID, CLID)

2.On receiving shuttle from previous replica:
executeOperation(O, UID, CLID, shuttle)

3.On receiving result shuttle from next replica:
updateHistory(result_shuttle)

4. On receiving “wedge” message from Olympus:
sendWedgedStatement()

```
FUNCTION: executeOperation(O, UID, CLID, …)                    #overloaded function
SWITCH:
    CASE replica is HEAD:
        #check if <O, UID, CLID> exists in HISTORY
        #if YES, then it is a retransmission
        #if NO,  then it is a normal execution

        ret_val = check_in_history(O, UID, CLID)
        if ret_val == NO:                                      #<O,UID,CLID> does not exist in
                                                               HISTORY
            s = create_new_slotnumber()
            shuttle = create_shuttle()                         #initialized shuttle with <s,O> and
                                                               empty order_proof[] and result_proof[]
            shuttle.order_proof.hash_array.add(HASH( [s,O], private_key_of_replica ))
                                                               #HASH(<set>, KEY)
            result = execute_operation_on_object()             #perform the operation O on current
                                                               state of object
            shuttle.result_proof.hash_array.add(HASH([r], private_key_of_replica))
            HISTORY.add(s, O, CONFIG_ID, UID, CLID)
            HISTORY.addOrderProof(s, O, HASH( [s,O], private_key_of_replica ))
                                                               #Add order proof for <s,O> in HISTORY
            send_shuttle_to_next(replica.getNext(), s, O, UID, CLID)

        else if ret_val == YES:                                #Retransmission due to client timeout
            check_if_operation_in_history(s, O, UID, CLID)
            if found in HISTORY:
                check_if_result_cached_in_history(s, O)        #Checks for result for <s,O>
                if found in HISTORY:                           #CASE 1 on PG. 9 of BCR paper
                    return history.r and history.result_proof
                else                                           #CASE 2 on PG. 9 of BCR paper
                    wait_for_result_shuttle()
                    if received result_shuttle:
                        return history.r and history.result_proof
                    start_timer()
                    if timer.time > TIMEOUT                    #TIMEOUT
                        becomeImmutable()
                        send_reconfig_request_to_olympus()
            else                                               #CASE 3 on PG.9 of BCR paper
                executeOperation(O, UID, CLID)
                start_timer()
                if timer.time > TIMEOUT                        #TIMEOUT
```

```
                    becomeImmutable()
                    send_reconfig_request_to_olympus()

CASE (replica is in between HEAD and TAIL):
    ret_val = check_in_history_if_valid(s, O)          #check order proof for <s,O> in history
    if ret_value == yes                                #NO misbehavior
        ret_val = check_in_history(O, UID, CLID)       #check if execute message was received
                                                       as a part of broadcast on client timeout
        if ret_val == NO
            shuttle.order_proof.hash_array.add(HASH( <s,O>, private_key_of_replica ))
            result = execute_operation_on_object()
            shuttle.result_proof.hash_array.add(HASH(r, private_key_of_replica))
            HISTORY.add(s, O, CONFIG_ID, UID, CLID, r)
            send_shuttle_to_next(next_replica, s, O, UID, CLID)

        else if ret_val == YES                         #received due to client retransmission
            check_if_operation_in_history(s, O, UID, CLID)
            if found in HISTORY:
                check_if_result_cached_in_history(s, O)
                                                       #Checks for result for <s,O>
                if found in HISTORY:                   #CASE 1 on PG. 9 of BCR paper
                    return history.r and history.result_proof
                else                                   #CASE 2 on PG. 9 of BCR paper
                    wait_for_result_shuttle()
                    if received result_shuttle:
                        return history.r and history.result_proof
                    start_timer()
                    if timer.time > TIMEOUT
                        becomeImmutable()
                        send_reconfig_request_to_olympus()
            else                                       #CASE 3 on PG.9 of BCR paper
                send_operation_to_head(O, UID, CLID)
                start_timer()
                    if timer.time > TIMEOUT
                        becomeImmutable()
                        send_reconfig_request_to_olympus()


    else if boolreply == NO:                           #same slot different operation
                                                       i.e. detected MISBEHAVIOR
        becomeImmutable()
        send_reconfig_request_to_olympus()

CASE (replica is TAIL):
    ret_val = check_in_history_if_valid(s, O)          #check order proof for <s,O> in history
    if ret_value == yes                                #NO misbehavior
        ret_val = check_in_history(O, UID, CLID)       #check if execute message was received
                                                       as a part of broadcast on client timeout
        if ret_val == NO
            shuttle.order_proof.hash_array.add(HASH( <s,O>, private_key_of_replica ))
            result = execute_operation_on_object()
            shuttle.result_proof.hash_array.add(HASH(r, private_key_of_replica))
            HISTORY.add(s, O, CONFIG_ID, UID, CLID, r)
            result_shuttle = create_result_shuttle(s, r, UID, CLID, shuttle.resultproof)
            send_to_client(r, shuttle.result_proof)
            send_back_result_shuttle(r, result_shuttle) #send completed result proof in result
                                                       shuttle
```

```
        else if ret_val == YES                          #received due to client retransmission
                check_if_operation_in_history(s, O, UID, CLID)
                if found in HISTORY:
                        check_if_result_cached_in_history(s, O)
                                                        #Checks for result for <s,O>
                        if found in HISTORY:            #CASE 1 on PG. 9 of BCR paper
                                return history.r and history.result_proof
                        else                            #CASE 2 on PG. 9 of BCR paper
                                wait_for_result_shuttle()
                                if received result_shuttle:
                                        return history.r and history.result_proof
                                start_timer()
                                if timer.time > TIMEOUT
                                        becomeImmutable()
                                        send_reconfig_request_to_olympus()
                else                                    #CASE 3 on PG.9 of BCR paper
                        send_operation_to_head(O, UID, CLID)
                        start_timer()
                                if timer.time > TIMEOUT
                                        becomeImmutable()
                                        send_reconfig_request_to_olympus()


    else if boolreply == NO:                            #same slot different operation
                                                        i.e. detected MISBEHAVIOR
            becomeImmutable()
            send_reconfig_request_to_olympus()
```

```
FUNCTION: updateHistory(result_shuttle)
HISTORY.add_result(result_shuttle.r, result_shuttle.s)      #check for slot s and enter result r
HISTORY.add_result_proof(result_shuttle.result_proof, result_shuttle.s)
                                                            #check for slot s and enter result_proof
                                                            NOTE:this is completed result_proof
```

```
FUNCTION: sendWedgedStatement()
wedged_statement = createNewWedgedStatement()
wedged_statement.add(self.HISTORY.orders_proofs)
wedged_statement.add(self.HISTORY.checkpoint_proofs)
```

```
FUNCTION: addToCheckPointProof(slot):
If replica is HEAD:
    shuttle = create_checkpoint_shuttle()           #initialize a shuttle with empty
                                                    checkpoint_proof and slot_number
                                                    NOTE: slot_number also acts as
                                                    checkpoint_id, HASH are added indexed
                                                    at replica_id
shuttle.checkpoint.proof(HASH(state, private_key_of_replica))

if replica is TAIL:
    send_shuttle_back_chain()                       #shuttle goes in reverse direction in
                                                    chain with completed checkpoint
                                                    proofs
```

OLYMPUS

1.On receiving reconfig_statement from replica:

reconfigure()

FUNCTION: **reconfige**()

```
for all replica in current_config:
    send_wedge_request()
    wait_for_wedge_statements()                          #waits for all wedged_statements to be
                                                         returned from replicas

    for all q in quorum_set:
        slot = get_latest_checkpoint_slot_number(wedged_statements)
        longest_h = get_longest_history(wedged_statements)
        result = check_if_history_is_consistent(slot, longest_h, wedged_statements)
        if result == -1
                continue;
        else                                             #found quorum with consistent replicas
                //catch up with longest replica
                //check if final states are consistent
                catchup_with_longest_replica(longest_h, q, wedged_statements)

                                                         #compares all replicas and sends 1 on
                                                         success, -1 on failure
                ret_val = check_if_all_final_results_are_same()
                if ret_val == 1:                         #all final states are consistent
                        get_final_state_from_any_replica_in_q()  #gets final state
                        exit(SUCCESS)                    #New Configuration has been created

                else                                     #final states are not consistent
                        continue;                         #select new quorum
```

FUNCTION: **get_latest_checkpoint_slot_number**(wedged_statements)

```
max = 0
for every checkpointproof in wedged_statements:
    if max < checkpointproof.slot
        max = checkpointproof.slot
    return slot
```

FUNCTION: **get_longest_history**(wedged_statements)

```
longest_h = NULL
longest_size = 0
for every history in wedged_statements:
    if longest_size < history.length
        longest_size = history.length
        longest_h = history
    return longest_h
```

FUNCTION: **check_if_history_is_consistent**(slot, longest_h, wedged_statements)

```
for all history in wedged_statements:
    for slot_number in range (slot, history.highest_slot):
        if history.operation_for_slot(s) == longest_h.operation_for_slot(s)
```

```
            continue
        else                        #either replica might be faulty-> choose different quorum
            return -1
```

```
FUNCTION: catchup_with_longest_replica(longest, q, wedged_statements):
for all replica in q:
    offset = get_offset_operations(replica, longest, wedged_statements)
    final_state_array[replica_id] = catchup(replica, offset)
                                                    #final_state_array contains a hash ch
                                                    (running state of each replica hashed
                                                    with public key of client)
                                                    store this has as CH
```

```
FUNCTION: get_offset_operations(replica, longest, wedged_statements)
offset = []
for all slots in longest:
    if !wedged_statement[replica].history.has(slot)     #SLOT doesn't exist in history of replica
    offset.add(slot, longest.operationAt(slot))

return offset
```