Byzantine Chain Replication Implementation

Part 1 : Pseudo code

Submission for CSE 535 Course Project

Team members

Saraj Munjal
NetId: smunjal
ID #: 111497962
Email: smunjal@cs.stonybrook.edu

Ankit Agarwal
NetId: anaaggarwal
ID #: 111485578
Email: anaaggarwal@cs.stonybrook.edu

Client Object

```
Client:
    # client state variables
    Olympus
    T = T                    # Max number of tolerated failures in system
    timeout = x seconds     # client's timeout for queries
    replicas = []            # replica IDs in the current configuration
    replica_public_keys = {} # mapping of replica ID vs its public key

    # client methods
     request_configuration():
        send('get_configuration') to Olympus
        wait(on_message_received('configuration'), timeout)


     on_message_received('configuration', replicas, replica_public_keys):
        replicas = replicas
        replica_public_keys = replica_public_keys


     generate_request_id(): returns random monotonic request ID, like timestamp, UUID

     execute_operation(o):
        # o is the operation that the client wants to execute
        request_configuration()
        # we assume that configuration is updated through Olympus
        request_id = generate_request_id()
        head = replicas[0]
        send('request', o, request_id) to head
        wait(on_message_received('result'), timeout)
        handle_request_retry(request_id, o)

     on_message_received('result', result, result_proofs, o):
        # On receiving a result, check if the result hash is correct for atleast (T + 1) replicas so that atleast one is honest
        # Assumption: result_proofs from each replica r' are in the same order as replicas
        verified_count = 0
        for i from 1 to size(replicas):
            if verify_signature(<o, hash(result)>, result_proofs[i], replica_public_keys[i]) == true:
                verified_count += 1

        if verified_count >= T + 1:
```

```
            log('Operation successful', o)
        else
            log('Operation failure', o)
            send('request_reconfiguration_client', result, result_proofs, o) to Olympus
        # client operation successful
        exit

     on_message_received('retry_error', o) {
        # This means that re-configuration is taking place in the system. Retry again after a while.
        wait (timeout)
        execute_operation(o)
    }


     handle_request_retry(request_id, o):
        # now we will retry our request by sending to all replicas in the chain, not just the head
        for i from 1 to size(replicas) in parallel:
            send('request', o, request_id) to replica[i]
            wait (on_message_received('result' || 'retry_error'), timeout)


     verify_signature(message, signature, key):
        # this Crypto function determines if the signature computed for message is valid with specified key
        # Standard implementations can be found in Crypto libraries
```

Olympus Object

```
Olympus:
    # Assumptions: generate_keys() is a function that initializes and returns a public-private key pair
    # Standard implementations can be found in Crypto libraries for most languages

    # Assumptions: wait_aggregate(N, func) is a function that waits till a nested function func is called n times
    # It aggregates all func operands as lists and returns them as tuples

    # Assumption: wait_for_message is a function that blockingly waits for a message and copies arguments into result

    # Assumption: hash utility function that calculates hash of the given data

    # Olympus state variables
    T                        # maximum number of failures that can be tolerated by system
    num_replicas = 2T + 1    # total replicas must be 2T + 1 to maintain consistency guarantees
```

```
replicas = []            # replica objects
replica_public_keys = [] # public keys of replicas
replica_private_keys = [] # private keys of replicas
replica_timeout

# Olympus constructor
Olympus (T_value, replica_timeout):
    T = T_value
    num_replicas = 2T + 1
    replica_timeout = replica_timeout


# Olympus methods


init():
    init_replicas([], [])  # since this is the first initialization of config, there is no history or state for replicas
    return replicas        # returns the replicas state variable


init_replicas(running_state, history):
    for i from 1 to num_replicas:
        <replica_public_keys[i], replica_private_keys[i]> = generate_keys()
        replica[i] = generate_replica(i, running_state, history, replicas, replica_public_keys, replica_private_keys[i], replica_timeout)


generate_replica(i, running_state, history, replicas, replica_public_keys, single_private_key):
    r = Replica(replicas, replica_public_keys, single_private_key, replica_timeout)
    replicas[i] = r
    r.state = ACTIVE
    send('init_hist', running_state, history) to r

# This is the message received by Olympus from clients asking for configuration
on_message_received('get_configuration', requester):
    send('configuration', replicas, replica_public_keys) to requester


on_message_received('request_reconfiguration_client', result, result_proofs, o):
    # when Olympus receives an external reconfiguration request, i.e. from the client, it needs to verify
    # the proofs sent for reconfiguration. Hence, we do a signature verification for (T+1) result_proofs again
    verified_count = 0
    for i from 1 to num_replicas:
        if verify_signature(<o, hash(result)>, result_proofs[i], replica_public_keys[i]) == true:
```

```
            verified_count += 1

        if verified_count >= T + 1:
            # Reconfiguration request is invalid
            log ('Neglecting reconfiguration request', o)
            return
        begin_reconfiguration()


    begin_reconfiguration():
        for i = 1 to num_replicas in parallel:
            send('wedge_request') to replica[i]

        <histories, checkpoint_proofs, hashed_running_states> =
        wait_aggregate(num_replicas, on_message_received('wedge', history, checkpoint_proof, hash_running_state))
        # now we make at most T quorum attempts, each with a size of T + 1, and attempt to find a valid history
        for i from 1 to T:
            quorum_range = range (i to i + T + 1)
            quorum_result_pair = is_valid_quorum(histories (i, i + T + 1), quorum_range)
            if quorum_result_pair.first == true:
                # now, we need to get a valid running state from our quorum, and match it against the hash we got while checking validity
                for i in quorum_range:
                    send('get_running_state') to replica[i]
                    rs = wait_for_message('running_state', rs)
                    if (hash(rs) == quorum_result_pair.second):   # this means that the running state hash computed earlier while validating quorum,
                                                                  # matches the one sent by the currently queried replica. This means we can now reconfigure.
                        init_replicas(rs, lh)
                        return
        return


    # function that returns a tuple of boolean validity of quorum and hash of result
    is_valid_quorum(histories, indices):
        for each <i,j> in indices:                      # proceed for each pair <i,j> in indices, nested loop
            for <si, oi> in histories[i]:               # for each tuple <si,oi> in history[i]
                <sj, oj> = histories[j].find(<si, *>)  # find a tuple <sj, oj> in history[j] such that s = si, o = anything
                if oi != oj:                            # if oi and oj don't match, the history is invalid due to a possible failure
                    return <false, null>
        # the quorum is valid up to this point. Now we ask each replica to catch up.
        lh = max(histories)
        for i in indices:
            send('catch_up', lh - histories[i]) to replica[i]
        hashed_running_states = wait_aggregate(size(indices), on_message_received('caught_up', hashed_running_state))
```

```
        unique_running_states = unique(hashed_running_states)
        if  size(unique_running_state) > 1:
            # all hashes must be the same for a valid quorum
            return <false, null>
        return <true, unique_running_states[0]>


    on_message_received('request_reconfiguration_replica'):
        # this function is triggered when a replica triggers a reconfiguration
        # in this case, Olympus doesn't need to verify anything. It trusts the replica's request
        begin_reconfiguration()
```

Request Shuttle Object

```
Request_Shuttle:
    # Convention: s => monotonically increasing slot number for client request, assigned by head
    # Convention: o => operation to be performed as a part of client request

    # Assumption: we don't need to keep a separate slot member variable, replicas
    # handling request shuttle can automatically infer it from order_proofs

    # Assumption: according to BCR algorithm, order proof tuples also contain configuration C as a param.
    # However, we have omitted configuration C because we are not reusing replicas across configurations,
    # and hence configurations are independent and not required here

    # request shuttle state variables
    request_id: # Unique Identifier for client request: can be timestamp, UUID etc
    order_proofs = []  # list of tuple signatures:  Sign(<s,o, replica_id> using private key of replica_id)
    result_proofs = [] # list of tuple signatures for the result at each replica that has processed shuttle
                       # : Sign(<o, hash(result)> using private key of replica_id)


    # request shuttle constructor
    Request_Shuttle(req_id):
        request_id = req_id
        order_proofs = []
        result_proofs = []
```

Result Shuttle Object

```
Result_Shuttle:
    # Convention: o => operation to be performed as a part of client request
    # Assumption: result of applying operation o to each replica is the same

    r # result after applying operation to the running state of the tail
    result_proofs = [] # list of tuple signatures for the result at each replica
                           #: Sign(<o, hash(result)> using private key of replica_id)
```

Replica Object

```
# Assumption: The slot number is global and not per client.
global slots_used = 0 # initialize global variable slot number

global get_slot_number():
    slots_used += 1
    return slots_used

Replica:
    # Assumption: One shuttle supports one operation per each client.
    # Assumption: Only one configuration is active at a given time.

    # A replica can have either of three states at a given time: ACTIVE, PENDING, or IMMUTABLE
    # Initial configuration has all replicas with active state
    # All other configuration replicas are in pending state
    # A replica goes to immutable state in case of a potential failure (upon wedge request from Olympus)
    # sign(data, key) => Sign the data with the given private key
    # verify_signature(message, signature, key) => Check private key's signature with a public key and message
    # hash => Calculate hash of the given data
    # s => slot number allotted to the requested operation
    # o => requested operation by the client
    # head => head replica of the current configuration
    # tail => tail replica of the current configuration
    # result_cache = {} # key-value store to cache result against particular request id
    # checkpoint_proof => list of hashed running states of each replica along the chain
    # T => number of faults that can be tolerated by 2T + 1 replicas
    # timer => creates a timer for the current replica
```

```
# replica state variables
replica_id  # id of the current replica
state = {ACTIVE | PENDING | IMMUTABLE}
running_state   # current state of the replica in the given configuration
history = []    # history of the current replica, contains the list of order proofs for all operations since the last checkpoint
private_key     # private key of current replica
replica_public_keys  # public keys of all replicas in the chain
timeout         # timeout period for operations

# replica constructor
Replica (replica_list, replica_public_keys, own_private_key, timeout):
    replicas = replica_list
    head = 0
    tail = 2T - 1
    replica_public_keys = replica_public_keys
    private_key = own_private_key
    timeout = timeout

# replica methods

send(event message, ...) to receiver:
    # sends the event message to the receiver with the given variable arguments
    # It is assumed that the programming language and compiler exposes an implementation of a similar method for
    # inter-process communication among different nodes


init_request(request_id, o): # head will start the request shuttle upon receiving client's request
    if replica_id == head:
        s = get_slot_number() # get the next unused slot number
        request_shuttle = Request_Shuttle(request_id) # initialize request shuttle
        wait(order_command(s, o, request_shuttle), timeout) # wait until the result_shuttle is received
        # Assumption: If the head receives the result shuttle before the timeout, timer will be cancelled


init_checkpointing():
    if slot_number / 100 is integer: # for every 100 slots, create new checkpoint and clear cache
        clear_cache() # clear cache after every 100 slots
        checkpoint_proof = [hash(running_state)] # initialize checkpoint proof
        checkpoint = slot_number
        send("checkpointing", checkpoint, checkpoint_proof) to replicas[replica_id + 1]


apply_operation(o):
```

```
    result = o(running_state) # running_state is modified to new running_state
    return result

is_valid_proof(s, o, order_proof):
    for each <s_p, o_p, r_p> in order_proof: # s_p, o_p, and r_p is the slot number, operation and replica id of each previous replica in the
                                         # configuration respectively
    if s == s_p and o != o_p:
        return false
    return true

order_command(s, o, request_shuttle):
    for each previous replica r_p:
        if not verify_signature(request_shuttle.order_proof of r_p, public key of r_p):
            send('request_reconfiguration_replica') to Olympus # notify Olympus if any of the previous replica is incorrectly signed
            return
    if not is_valid_proof(s, o, request_shuttle.order_proof):
        send('request_reconfiguration_replica') to Olympus # notify Olympus if the operation described against a specific slot number is not unique in the
                                                   # order-proofs of previous replicas
        return
    request_shuttle.order_proof = request_shuttle.order_proof + H([s, o], replica_private_key)
    history = history + [request_shuttle.order_proof]
    result = apply_operation(o)
    request_shuttle.result_proof = request_shuttle.result_proof + H([o, hash(result)], replica_private_key)
    return request_shuttle

# Assumption: instead of making multiple methods for receiving messages, we make one method that does if-else handling
on_message_received(statement, ...):
    if statement == "request": # request can be either from client or peer replicas
        send_result_to_client(request_id_args) # request_id_args => request_id_args passed from the variable arguments

    else if statement == "wedge": # On receiving wedge request from Olympus
        state = IMMUTABLE # change state to IMMUTABLE
        send("wedge_request", history, checkpoint_proof, hash(running_state)) to Olympus

    else if statement == "catch_up": # Upon receiving catch up request from Olympus
        catch_up(set_of_operations_to_apply)

    else if statement == "get_running_state":
        send("running_state", running_state) to Olympus

    else if statement == "result_shuttle":
        result_shuttle = receive_result_shuttle_from_successor(request_id, result_shuttle_args) # result_shuttle_args => result_shuttle passed from the
                                                                                         # variable arguments
```

```
            forward_result_shuttle_to_predessor(request_id, result_shuttle)

        else if statement == "request_shuttle":
            request_shuttle = receive_request_shuttle_from_predessor(s, o, request_shuttle_args)     # request_shuttle_args => request_shuttle passed from the
                                                                                                     # variable arguments
            forward_request_shuttle_to_successor(request_shuttle)

        else if statement == "checkpointing":
            if replica_id < tail:
                checkpoint_proof = checkpoint_proof_args + [hash(running_state)] # checkpoint_proof_args => checkpoint_proof passed from the
                                                                                 # variable arguments
                send('checkpointing', checkpoint, checkpoint_proof) to replicas[replica_id + 1]
            else:
                send('checkpoint_proof', checkpoint) to replicas[replica_id - 1]
                history = history[checkpoint:] # truncating history before the checkpoint

        else if statement == "checkpoint_proof":
            if replica_id < tail:
                send("checkpoint_proof", checkpoint) to replicas[replica_id - 1]
                history = history[checkpoint:] # truncating history before the checkpoint


    forward_result_shuttle_to_predessor(request_id, result_shuttle):
        send("result_shuttle", request_id, result_shuttle) to replicas[replica_id - 1]


    receive_result_shuttle_from_successor(request_id, request_shuttle):
        result_cache[request_id] = (result_shuttle.r, result_shuttle.result_proof) # result_shuttle.r => result provided by predecessor


    forward_request_shuttle_to_successor(request_shuttle):
        send("request_shuttle", s, o, request_shuttle.request_id, request_shuttle) to replicas[replica_id + 1]


    receive_request_shuttle_from_predessor(s, o, request_shuttle):
        request_shuttle = order_command(s, o, request_shuttle)
        return request_shuttle


    check_cache(request_id): # check result cache for previously saved results
        if request_id in result_cache:
            return true
        return false
```

```
clear_cache():
    if slot_number / 100 is an integer:
        result_cache = {} # clear cache for every 100 slots, other cache management techniques can also
                          # be applied to enhance implementation performance


send_result_to_client(request_id, client):
    if check_cache(request_id):
        send('result', result_cache[request_id]) to client
    else:
        if replica_id > head:
            send('redirect_to_head', request_id) to replicas[head]  # redirect request to head if not found in result cache
            wait(on_message_received('result_shuttle'), timeout) # wait for result_shuttle message till timeout
            # if the request times out, we need to reconfigure the system
            send('request_reconfiguration_replica') to Olympus
        else:
            init_request(request_id) # start the request from scratch


catch_up(set_of_operations_to_apply):
    for each operation o_c in set_of_operations_to_apply:
        result = o_c(running_state) # running_state will be modified to the new_running_state
    send('caught_up') to Olympus # send a caught_up message to Olympus after catching up with the latest running state
```