Byzantine chain replication

- Nirvik Ghosh(nghosh)
- Nihal Harish (nharish)

Roles:
1. Client
2. Replicas
3. Olympus

Some definition's that will be used throughout are:
encrypt(data, p_key) is defined as encrypting data with some key called p_key
decrypt(data, p_key) is defined as decrypting data with some key called p_key
SHA256(data) is defined as encrypting data with SHA256

CLIENT

Local State of Client:

1. **client_id**: UUID
2. **private_key**: given by Olympus
3. **public keys of all replicas**: given by Olympus
4. **public key of the olympus**: given by Olympus
5. **request number**: 0
6. **timer_request**: timer value which triggers when it sends a new request
7. **timer_retransmission**: timer value which triggers when it retransmits a request
8. **cached_config_file**: it will be empty intially. It contains the configuration file from the Olympus
9. **set_of_operations**: a set of operations client can perform on the object

Methods:

1. **request**
2. **retransmission**
3. **check_validity_of_result**

Client Messages:
Messages that client sends are as follows:
1. **operation**: This message is sent when the client wishes to initiate an operation. The message format

```
{
        type = “operation”
        body = {
           client_id : <id of the client>,
           signed_request : encrypt((operation, client_id, request_number), client.private_key)
        }
}
```

2. **retransmission**: This message is sent when client's request was not answered within specific time. The message format is same as the operation message format, except the “type” field.

```
{
```

type = “retransmission”,
body = {
client_id : <id of the client>,
signed_request : encrypt((operation, client_id, request_number), client.private_key)
}
}

3. **get_latest_config**: This message is sent to the Olympus, to get the latest configuration details. The message format is

{
type = “get_latest_config”,
body = {
client_id : <id of the client>,
nonce : <random_value>
signature: encrypt(nonce, client.private_key)
}
}

4. **proof_of_misbehavior**: This message is sent to the Olympus to inform that the result proof received is inconsistent and that Olympus should look into this matter. The message format is as below

{
type = “proof_of_misbehavior”,
body = {
result_proof : <result_proof>
order_proof : <order_proof>
}
}

NOTE: sender can send either result_proof or order_proof. If sender sends result_proof, then order_proof will be NULL and vice versa.

Messages that client receives are as follows:

1. **result**: This message is initiated by the tail replica or multiple replicas.

The format of the message ->

{
type = “result”,
body = {
result : _,
result_proof : []
}
}

2. **error**: This message is initiated by an immutable replica. The format of the message is

{
type = “error”
}

Request:
This method is responsible for following actions:
1. Save and fetch new configuration details (if config is saved, do not fetch)
2. make request of some operation to the head replica

3. if response takes too long, then invalidate the configuration details and repeat 1.
4. else if response is received, then check the validity of the response
    4.1 if response contains an 'error' field, this implies that tail replica is immutable.
        4.1.1 invalidate the configuration details and repeat 1.
    4.2 if the response contains valid result_proof then success
    4.3 else the response is not valid, then send reconfigure message to Olympus, invalidate config details and repeat 1.

---

client.request()

```
if client.cached_config_file is Empty:
    client.cached_config_file  = get config details from olympus and save it
endif
client.timer_request = start_timer()
client.set_of_operations = Set of operations that a client can perform
client.request_number = select a new request number
```

client chooses an **operation** from the **set of operations** and makes an 'operation' request to the head which says ('operation', encrypt((operation, client_id, request_number) , client.private_key))

```
if client.timer_request expires then
    client.retransmission(request_number)        // retransmit the same requeste number
else if response received before client.timer_request expires then
    decrypt the response
    if response.type == 'result' then
        validity_message = check_validity_of_result(response.body)
        if validity_message == 'success' then          //  result proof is validated
            success and return
        else if validity_message == 'error' then    // result proof not signed by t+1 replicas or an immutable replica has replied
            invalidate the client.cached_config_file details
            client.request()
        endif
    else if response.type == 'error' then  // tail becomes immutable and send 'error'
        invalidate the client.cached_config_file details
        client.request()
    endif
endif
```

Retransmission: This method retransmits the operation to all the replicas. If the result is received before expiry, the validity of result is checked and appropriate actions are taken. Else if timer expires then current config details invalidated and new configuration file is requested.

1. Start the timer_retransmission
2. Send retransmission message to all replicas where a signed message: (same operation, client_id, request_number) and an unencrypted client_id is a part of the retransmission message.
    2.1 To check if client is valid, the replica can compare the unencrypted client_id with the decrypted client_id in the signed message.

3. Check for response
4. There will two situations
    4.1 replica responds with an 'error' message if its immutable
        4.1.1 client invalidate the configuration details and make a new config request to Olympus
    4.2 replicas respond with the result shuttle and client validates it to check if it can accept the result or it has to get a new configuration file from the olympus

---

client.retransmission(request)

```
client.timer_retransmission = start_timer()
for each replica in set_of_replicas do
    send retransmission_message('retransmission', client_id, encrypt((operation, client_id,
                                same_request_number), client.private_key)) to replica

if response is received before client.timer_retransmission expires then
    decrypt the response
    if the response.type == 'error' then     // immutable object replies
        invalidate the client.cached_config_file details
        client.request()
    else
        validity_message = check_validity_of_result(response)
        if validity_message == 'success' then
            success and return
        else if validity_message == 'error' then
            invalidate the client.cached_config_file details
            send('proof_of_misbehavior', response) to Olympus
            client.request() after some time
    endif

else if the timer has expired then
    invalidate the client.cached_config_file details
    client.request()
endif
```

---

check_validity_of result (response): This method checks if the “result_proof” has been signed by atleast t+1 replicas and the result is consistent.

---

client.check_validity_of_result(response)

```
extract { result, complete_result_proof } from response
hashed_result = SHA256(result)
count = 0
for each (index, proof) in enumerate(result_proof):
    if hashed_result == decrypt(proof, public key of replica[index]) then
        increment count
    endif
if count > t+1 then
```

```
            return success
        endif
        return failed
```

Replica:

Local State of Replica:

1. **running_state**: a dictionary object (initial state: Empty)
2. **history**: [<slot, operation, replica_number, config_number, order_proof_slot>_replica, ……] (Initial state: Empty)
3. **mode**: PENDING | ACTIVE | IMMUTABLE
4. **next_replica**: next replica address
5. **previous_replica**: previous replica address
6. **public_keys**: dictionary that contains public keys of all replica, client, olympus
7. **private_key**: Its own Private key
8. **result_shuttle_cache**: store result_shuttle
9. **checkpoint_proof_cache**: store complete checkpoint proof
10. **isHead**: if the replica is head or not
11. **isTail**: if the replica is tail or not
12. **timer**: used during retransmission
13. **current_slot** : contains the current slot number
14. **config_number**: configuration number it belongs to
15. **retransmission**: It is a list of client's retransmitted request checksums
16. **waiting_for_result_proof:** It is store which informs on which complete_result_proof it is waiting for

Methods:

1. **run**
2. **receive_handler**
3. **handle_operation**
4. **generate_shuttle_message**
5. **handle_shuttle**
6. **check_order_proof_validity**
7. **check_result_proof_validity**
8. **check_checkpoint_proof_validity**
9. **handle_result_shuttle**
10. **handle_retransmission**
11. **handle_catchup_request**
12. **handle_checkpoint**
13. **handle_checkpoint_proof**
14 **handle_get_running_state_request**

There will be special handlers for Head and Tail.

Run:
This method will be running throughout the lifetime of the replica. It listens for messages and calls the receive_handler method. It also triggers the checkpoint shuttle when the current slot number becomes a multiple of 10.

---

replica.run()

```
message = listen_messages()
receive_handler(message)
if replica isHead then
    if current_slot_number is multiple of 10 then
        hashed_state = SHA256(running_state)
        checkpoint_msg = (hashed_state and current_slot_number)
        send ('checkpoint', checkpoint_msg) to replica.next_replica
    endif
endif
```

---

receive_handler:
This method is responsible for dispatching the appropriate handlers depending on the message.
The following are the **message types and formats** which are expected to be received at replica:

1. '**retransmission**': This message comes from the client when the client's timer expires.
   The message format looks as follows:

```
{
    type: “retransmission”,
    body: {
        client_id: <client_id>;
        signed_request: encrypt((operation, client_id, request_number),
                                client_private_key)
    }
}
```

2. '**operation**': This message comes from the client when it wishes to make a request
   The message format looks as follows:

```
{
    type : “operation”,
    body : {
        client_id: <client_id>,
        signed_request: encrypt((operation, client_id, request_number),
client_private_key)
    }
}
```

3. '**wedge_request**': This message comes from the Olympus, when there is a re-configuration stage running in Olympus. This message is signed by the Olympus, so that it prevents the replicas from random 'wedge_request'. The message format looks as follows:

```
{
    type : “wedge_request”,
```

body: {
nonce : <random_value>,
signed_message: encrypt(nonce, private key of Olympus)
}
}

4. '**shuttle**': This message is initiated by the head and is forwarded to the next replica until it reaches tail

The message format looks as follows:

{
type = 'shuttle',
body = {
'result_proof': [_] ,
'order_proof': [_] ,
'slot': _,
'operation': _ ,
'checksum': _,
'client_id': _
}
}

5. '**result_shuttle**': This message is sent by the tail and is propagated till the head. The message format is similar to shuttle message, however, it contains one new field called 'result' and new slightly new field names.

{
type = 'result_shuttle',
body = {
'complete_result_proof': [_] ,
'complete_order_proof': [_] ,
'result': _,
'slot': _,
'operation': _ ,
'checksum': _,
'client_id': _
}
}

6. '**checkpoint**' : This message is initiated by the head for the purpose of coming to mutual consensus of truncating history before specified slot. The message format is as below

{
type = 'checkpoint',
body = {
slot: <slot_number>,
checkpoint_proof: [signed hash(running_state) of preceding replicas]
}
}

7. '**checkpoint_proof**': This message is initiated by the tail. When this message arrives, the replicas read the slot number from the message and remove all history before that slot number.

{
type = “checkpoint_proof”,
body = {
slot : <slot_number>,

```
                complete_checkpoint_proof: [ signed hash(running_state) of all replicas ]
                result : SHA256(running_state)
        }
}
```

8. '**catchup_message**': This is a message sent by the Olympus, requesting the replicas to perform a set of operations on the running_state. The message is signed because it prevents the replicas from getting random 'catchup_messages'. The message format is as follows:

```
{
        type = “catchup_message”,
        body = {
                signed_message : encrypt(operations, olympus.private_key)
                hash_of_operations : SHA256(operations)
        }
}
```

9. '**get_running_state**': This message is sent by the Olympus to the replica in order to retrieve the state of the object. This message is signed as well by the Olympus, to avoid malicious nodes to retrieve the current state of object. The message format is as below:

```
{
        type = “get_running_state”,
        body = {
                nonce: <random_value>
                signed_message: encrypt(nonce, olympus.private_key)
        }
}
```

---

replica.receive_handler(message):

```
if message.type == “retransmission” then                    // from client or from replica
        if message has arrived from client or replica then  // client could forward req to head
                replica.handle_retransmission(message.body)
        else
                drop the request and ignore
        endif

else if message.type == 'operation' then                    // from client
        if message has arrived from client then
                replica.handle_operation(message.body)
        else
                drop the request and ignore
        endif

else if message.type == 'wedge_request' then                // from olympus
        if message has arrived from Olympus then
                replica.handle_wedge_request(message.body)
        else
                drop the request and ignore
        endif

else if message.type == 'result_shuttle' then               // from replicas
        if message has arrived from replica.next_replica then
```

```
                replica.handle_resultShuttle(message.body)
            else
                drop the request and ignore
            endif

        else if message.type == 'shuttle' then                  // from replicas
            if message has arrived from replica.previous_replica then
                replica.handle_shuttle(message.body)
            else
                drop the request and return
            endif

        else if message.type == 'checkpoint' then               // from replicas
            if message has arrived from replica.previous_replica then
                replica.handle_checkpoint(message.body)
            else
                drop the request and return
            endif

        else if message.type == 'checkpoint_proof' then         // from replicas
            if message has arrived from replica.next_replica then
                replica.handle_checkpoint_proof(message.body)
            else
                drop the request and return
            endif

        else if message.type == 'catchup_message' then          // from Olympus
            replica.handle_catchup_request(message.body)

        else if message.type == 'get_running_state' then        // from Olympus
            replica.handle_get_running_state_request(message.body)
        endif
```

---

handle_operation: This method is responsible for handling requests from client. This doesn't take retransmission message. Only the head takes this request and creates a shuttle.

1. Checks if the request came from an authentic client
2. Get the request number and operation after decrypting the message
3. Compute the result and create result_proof
4. Add to order proof
5. Add <slot, operation, replica, config_number, order_proof> to its history
6. compute a **checksum** and wait for its complete order proof
7. Send the shuttle to the next_replica

checksum:
In this method, we compute the checksum as SHA256 of (client_id + request_number)
This checksum will be used as a key for expressing that its waiting for a complete result proof on that

request number and client id. This can be used during failure cases, replicas can determine if the operation has already been performed and is waiting for the result proof to arrive.

---

replica.handle_operation(message)

```
    if isHead then
        extract { client_id, signed_request } from the message
        extract client_id` from decrypt(signed_request, public_key[client_id])
        if (signed_request cannot be decrypted) or (client_id` != client_id) then
            drop the request and return
        endif
        extract { request_number, operation } from decrypted signed_request
        slot = get new slot number
        apply operation to the running state

        result = current value of the running state
        result_hash = compute SHA256(result)
        result_proof = [ sign the (result, result_hash) with replica.private_key ]

        order_proof = [ sign the (order, slot, opr) with replica.private_key ]
        add (slot, operation, replica, config_number, order_proof) to replica.history

        checksum = SHA256(client_id, request_number)
        shuttle_message = replica.generate_shuttle_message(result_proof, order_proof, slot,
operation, checksum, client_id)

        waiting_for_result_proof[checksum] = true
        send (type='shuttle', body=shuttle_message) to replica.next_replica
    endif
```

---

generate_shuttle_message: This method is responsible for generating the shuttle message. We need the client_id as well, so that the tail can figure out whom to send the response after computing the result. client_id parameter can also help the tail to sign the final response with appropriate client public key.

---

replica.generate_shuttle_message(result_proof, order_proof, slot, operation, checksum, client_id)

```
    shuttle_message = {
        'result_proof': result_proof,
        'order_proof': order_proof,
        'slot': slot,
        'operation': operation,
        'checksum': checksum,
        'client_id': client_id
    }
```

```
    return shuttle_message
```

---

handle_shuttle: This method is called when the replicas receive a shuttle message from their previous replica.
In this case, the replica determines the validity of the order proof.
On success, it computes the result and adds signed (hash(result)) statement to the result proof.
It also adds its signed order statement to the order proof.
The order statement is also added to the replica's history.
It uses the checksum to show that it is waiting for the complete result proof. This was explained earlier.
Then the shuttle is forwarded to the next replica.
If the replica itself is tail, it forwards the (result, result_proof) statement to the client and forward complete_result_proof to previous replica.

```
replica.handle_shuttle(shuttle_message)
    extract {result_proof, order_proof, slot, operation, checksum, client_id} from shuttle_message
    if the slot doesn't exist in its replica.history then
        replica.check_order_proof_valdity(slot, operation, order_proof)
        if order_proof is invalid then
            change replica.mode to IMMUTABLE
            send ('proof_of_misbehavior', (order_proof, slot, operation)) to Olympus

        else if order_proof is valid then
            apply operation to replica's running_state
            order_proof = order_proof U [  sign the (order, slot, operation) with
replica.private_key ]
            result = current running_state
            result_hash = SHA256(result)
            result_validity = replica.check_result_proof_validity(result_hash, result_proof)

            if result_validity == 'success' then
                result_proof = result_proof U [  sign the (result, result_hash) with
                                                    replica.private_key ]
                replica.history = replica.history U (slot, operation, replica,
                                                    config_number, order_proof)

                if replica isTail then
                    send ('result', (result; result_proof)) to client
                    result_shuttle = replica.generate_result_shuttle_message(
                                        slot,
                                        operation,
                                        result_proofs,
                                        order_proofs,
                                        checksum,
                                        client_id,
                                        result
                                    )
```

```
                                        send('result_shuttle', result_shuttle) to replica.previous_replica
                                    else if replica is not Tail then
                                        shuttle_message = replica.generate_shuttle_message(result_proof,
order_proof, slot, operation, checksum)
                                        waiting_for_result_proof[checksum] = true
                                        send ('shuttle', shuttle_message) to replica.next_replica
                                    endif

                                else if result_validity is inconsistent then
                                    send('proof_of_misbehavior', (result,result_proof)) to Olympus
                                endif
                            endif
                        else if slot exists in its history then
                            change replica.mode to IMMUTABLE
                            send_reconfigure_message_to_olympus()
                        endif
```

---

check_order_proof_valdity: This method is used for validating if the order proof contains no conflicting operations for a particular slot

---

```
replica.check_order_proof_valdity(slot, operation, order_proof)
        scan through the operations for a particular slot in the order_proof
        if all operations are same in that slot:
                return success
        else:
                return error
        endif
```

---

check_result_proof_validity: This method is used for determining the validity of result proof. It basically checks if all the result hashes are the same or not.

```
replica.check_result_proof_validity(result_hash, result_proof)
        check if all hashes in the result_proof are same
        if they are same then
                return success
        else then
                return error
        endif
```

---

handle_resultShuttle: This method is called when replica receives result shuttle from the next replica. We store the result_shuttle_message in result_cache[checksum], in case the client retransmits the same request.

If the checksum of (client_id + request_number) is present in the replica.retransmission, then forward the result proof immediately to the client.
Keep forwarding the result_shuttle to the previous replica until head is reached

---

replica.handle_resultShuttle(result_shuttle_msg)

```
    extract {complete_result_proof, complete_order_proof, slot, operation, checksum, client_id, result} from  result_shuttle_msg
    replica.result_cache[checksum] = result_shuttle_msg
    remove checksum from waiting_for_result_proof

    if  checksum in replica.retransmission then
        remove checksum from  replica.retransmission
        stop the timer_retransmission of that replica
        extract {result, complete_result_proof} from result_shuttle_msg
        send ('result', (result, result_proof)) to client
        if replica not isHead then
            send ('result_shuttle', result_shuttle_msg) to replica.previous_replica
        endif
    else:
        if replica is not Head then
            send ('result_shuttle', result_shuttle_msg) to replica.previous_replica
        endif
    endif
```

---

handle_retransmission: This method is for handling retransmission messages.
1. If a retranmission message arrives, the authenticity of the client is validated
2. While receiving result_shuttle, we maintained a result_cache where we store the result_shuttle corresponding to the checksum.
3. If we receive the same retransmission request, we query the result_cache and return it to the client immediately.
4. If the result_cache for that checksum is empty, we start the timer and forward the request to the head.
    4.1 When the replica is head, it checks if it has cached the result_shuttle for the corresponding checksum and returns the shuttle if true
    4.2 Else, it starts the timer. Then it checks if its already waiting for its result shuttle. If not, it creates a fresh new slot number and starts a new shuttle in the chain.
5. If timer expires before the result arrives, then reconfigure message is sent to the Olympus

---

replica.handle_retransmission(message)

```
    if (signed_request cannot be decrypted or client_id is not valid) then
        drop the request and return
    extract { operation, request_id, client_id } from decrypted signed_request

    checksum = SHA256(client_id, request_id)
```

```
        if checksum in replica.result_cache then
                extract { result, complete_result_proof } from replica.result_cache
                send(result, (result, complete_result_proof)) to client
                remove checksum from replica.result_cache          // No longer required
                return

        add checksum to replica.retransmission // so that when the result_shuttle arrives, replica can
return the answer directly. Refer handle_resultShuttle.

        else if replica.mode == 'IMMUTABLE' then
                remove checksum from replica.retransmission
                send(error) to client_addr

        else if replica is not Head then
                timer_retransmission = start the timer
                send ('retransmission', message) to head
                if t1 expires then
                        remove checksum from replica.retransmission
                        send_reconfigure_message_to_olympus()
                endif

        else if replica is Head then
                timer_retransmission = start the timer
                if not waiting_for_result_proof[checksum] then
                        remove checksum from replica.retransmission
                        replica.handle_operation(client_info, operation)  // Head calls the normal
operation handler, where it increases the slot number and creates a new shuttle
                endif
                if timer_retransmission expires then
                        remove checksum from replica.retransmission
                        send_reconfigure_message_to_olympus()
                endif
        endif
```

---

handle_wedge_request: This method is called when the Olympus sends a wedge request to the replica.
The sender is validated first.
After validation, the replica changes its mode to IMMUTABLE
Then it sends a signed wedge response to the Olympus containing (history, checkpoint_proof)

```
replica.handle_wedge_request(message_from_olympus)
        extract { nonce, signed_request } from message_from_olympus  // validating sender is Olympus
        if decrypt(signed_request, public_key_olympus) == nonce then
                replica.mode = IMMUTABLE
                wedge_statement = sign (replica.checkpoint_proof_cache, replica.history) with
                                        replica.private_key
```

```
        send ('wedge_response', wedge_statement) to Olympus
    endif
```

---

handle_catchup_request: This method is triggered when the Olympus sends a catchup request to the replicas.
The replica first validates the sender
if sender is validated, it extracts the set of operations and applies it on its running state
Then the hash of its running state is computed and sent to the Olympus

---

```
replica.handle_catchup_request(message_from_olympus)
    extract { hash_of_operations, signed_request } from message_from_olympus
    set_of_operations = decrypt(signed_request, public key of Olympus)
    if hash_of_operations == SHA256(set_of_operations) then
        for each operation x in set_of_operations do
            apply x on replica.running_state
        send ('caughtup', encrypt(SHA256(replica.running_state), replica.private_key)) to
        olympus
    endif
```

---

handle_checkpoint: This message is initiated when replica receives a checkpoint message from its previous replica.
On receiving the checkpoint message from previous replica, the checkpoint proof is validated (as described later)
Once the checkpoint proof is validated, the replica adds its checkpoint proof and forwards it to the next replica
If the tail receives the checkpoint, it truncates its history and propagates the complete_checkpoint_proof to its previous replica. The messages keeps going back until it reaches head.
If checkpoint is not validated, then a reconfigure message is sent to the Olympus

---

```
replica.handle_checkpoint(checkpoint_message)
    checkpoint_validity_message =
                        replica.check_checkpoint_proof_validity(checkpoint_message)
    if checkpoint_validity_message == 'success' then
        checkpoint_slot_number = get the slot number from the checkpoint_message
        checkpoint_proof = checkpoint_proof U [  sign the (SHA256(running_state),
checkpoint_slot_number) with replica.private_key ]

        if replica is not Tail then
            send checkpoint_proof to replica.next_replica
        else if replica is the Tail then
```

remove everything before checkpoint_slot_number from replica.history

send ('checkpoint_proof', (slot; checkpoint_proof; SHA256(result))) to replica.previous_replica

else if the checkpoint_validity_message is inconsistent do
send_reconfigure_message_to_olympus()
endif

---

handle_checkpoint_proof: This message is initiated by the tail replica. Each replica receives this message from its next replica.
The replica validates if all the hashes in the `checkpoint complete proof` message are the same or not.
If the complete_checkpoint_proof is validated then
replica saves the checkpoint proof in its checkpoint_proof_cache
This checkpoint_proof_cache can be used later for wedge response
Then truncate all the history before the given slot in the checkpoint proof
forward the checkpoint proof to your predecessor replica until it reaches head.

---

replica.handle_checkpoint_proof(checkpoint_proof_message)

if hashes in all signed checkpoint_proof statements are same then
replica.checkpoint_proof_cache = checkpoint_proof_message
extract { checkpoint_slot_number } from checkpoint_proof_message
remove everything before checkpoint_slot_number from replica.history
if replica is not Head then
send ('checkpoint_proof', checkpoint_proof_message) to replica.previous_replica
endif
endif

---

handle_get_running_state_request: This message is sent by the Olympus for sending the current state of the object.
The replica validates the sender of the request
If sender is validated, it sends its current state to the Olympus

---

replica.handle_get_running_state_request(message)
extract { nonce, signed_message } from message
if nonce == decrypt(signed_message, public key of Olympus) //Validating if sender is Olympus
send ('running_state', encrypt(replica.running_state, private key of replica)) to Olympus

---

check_checkpoint_proof_validity: This method is responsible for validating the checkpoint proof.
It checks if all the hashes in the checkpoint_proof are same or not .

If there is even one incorrect hash, error is returned

---

replica.check_checkpoint_proof_validity(checkpoint_message)

```
check if all hashes in the signed checkpoint proof are the same or not
if all are same then
        return success
else
        return error
endif
```

OLYMPUS

Local State of Olympus:

1. **set_of_quorum** = []
2. **caught_up_response** = []
3. **wedge_response** = []
4. **hashed_state_for_later_reference** = _
5. **timer_for_caughtup** = _
6. **timer_running_state** = _
7. **private_key**
8. **public and private keys of all replicas**

Methods of Olympus:

1. **run**
2. **handle_reconfigure**
3. **handle_proof_of_misbehavior**
4. **handle_caughtup_message**
5. **handle_running_state**
6. **handle_wedge_response**
7. **select_quorum**
8. **find_longest_history**
9. **invalidate_and_select**
10. **validate_result_proof_misbehavior**
11. **validate_order_proof_misbehavior**
12. **reconfigure**
13. **message_handler**

---

run: This method runs throughout the lifetime of Olympus.
It listens for incoming messages and handles them

---

olympus.run()

```
message = listen_messages()
message_handler(message)
```

---

message_handler: This method is responsible for dispatching appropriate handlers, depending on the type of message received.
The Olympus receives the following kinds of messages:
1. **reconfigure**: This message is sent by the replicas. The message format is as follows:

```
{
    type = “reconfigure”
}
```

2. **proof_of_misbehavior**: This message could be sent by client or replicas. If misbehavior is proved, then the Olympus will reconfigure. Note that client can not directly send 'reconfigure' message. It must prove the misbehavior.

```
{
    type = “proof_of_misbehavior”
    body = {
        result_proof: <result_proof>
        order_proof: <order_proof>
    }
}
```

NOTE: sender can send either result_proof or order_proof. If sender sends result_proof, then order_proof will be NULL.

3. **caught_up:** This message is sent by the replicas. It contains the hash of their running state.

```
{
    type = “caughtup”
    body = {
        signed_message: encrypt(hash_running_state, replica's private_key)
    }
}
```

4. **running_state:** This message is sent by the replica. It contains the running state of the replica.

```
{
    type = “running_state”,
    body = {
        signed_message: encrypt(running_state, replica's private key)
    }
}
```

5. **wedge_response:** This message is sent by the replica. It contains history and checkpoint_proof. The message format is as follows:

```
{
    type = “wedge_response”
    body = {
        signed_message : encrypt(wedge_statement, replica's private key)
    }
}
```

---

olympus.message_handler(data)

```
    sender, message = data
```

```
if message.type == 'reconfigure'
        olympus.handle_reconfigure(sender)

else if message.type == 'proof_of_misbehavior'
        olympus.handle_proof_of_misbehavior(sender, message.body)

else if message.type == 'caught_up'
        olympus.handle_caughtup_message(sender, message.body)

else if message.type == 'running_state'
        olympus.handle_running_state(sender, message.body)

else if message.type == 'wedge_response'
        olympus.handle_wedge_response(sender, message.body)

endif
```

---

handle_reconfigure: This method is called by the replica which belongs to the current configuration. Note that, even client can trigger this handler by proving some misbehavior. Refer handle_proof_of_misbehavior

---

olympus.handle_reconfigure(sender)

```
if sender is a replica belonging to the current configuration then
        olympus.reconfigure()
```

---

reconfigure: This method initializes all its reconfiguration variables and makes a signed wedge request to all replicas.

---

olympus.reconfigure()

```
olympus.set_of_quorum = EmptySet
olympus.caught_up_response = EmptySet
olympus.wedge_response = EmptySet
olympus.hashed_state_for_later_reference = Empty
signed_wedge_request =  ('wedge_request', (nonce, sign nonce with olympus.private_key))
for each replica in the configuration do
        send signed_wedge_request to replica
```

---

handle_wedge_response: This method aggregates all the wedge_response from replicas and is later used for selecting t+1 quorums and figuring out the longest history.

---

olympus.handle_wedge_response(replica, response)

```
extract { signed_message } from response
wedge_statement = decrypt(signed_message, replica's public key)
```

```
        olympus.wedge_response = olympus.wedge_response U wedge_statement
        if len(olympus.wedge_response) >= t+1
                olympus.set_of_quorum = olympus.select_quorum()
                olympus.find_longest_history()
        endif
```

---

select_quorum(): This method is responsible for selecting quorum of t+1 replicas such that their history is consistent.
consistent history means that for any pair of replicas in the quorum, if slot is present in both the replica's, then it must have the same operation. Some replicas might have more order commands than others. Replicas near the head will have extra order commands. The rest need to update.

---

olympus.select_quorum()

```
        if olympus.set_of_quorum is already set then
                ignore and return
        endif
        return quorum of t+1 replica's from olympus.wedge_response such that histories are consistent
```

---

find_longest_history: This method is responsible for finding the longest history and then sending a catchup request to replicas with missing histories.
The missing history is computed in the pseudocode with the operation '**-**'
(**a-b is defined as**: removing all history elements less than equal to b(slot_number) from a, where a is history object and b is the slot number).

Scenario:
Lets say Head has 200 slots and tail has 100 slots.
Tail hasn't received the remaining operations yet. However, during checkpointing, tail truncates its history first.
Tail removes its 100 slots from history.
Therefore, Head sends the complete history and Tail sends the truncated history along with the checkpoint proof as wedge_response.
Note that the checkpoint proof contains the last slot number field as well. Therefore, while computing the history_difference, since Tails history is empty, we still have access to its last checkpoint slot number.
We perform **(Head.history – Tail.checkpoint.slot_number)** => Remove all the slot numbers from Head.history which are less than Tail's slot number in the checkpoint proof.
If Tail had some history, along with checkpoint, then we could have just performed:
(Head.history – Tail.history.last_slot_number => Remove all slot numbers mentioned in the tail's history as well as the slot numbers before that

---

olympus.find_longest_history()

```
        longest_history = select history of replica whose history is the longest
        for each replica in the olympus.set_of_quorum do
                if  replica.wedged_history is not Empty then
                        history_difference = (longest_history –  replica.wedged_history.last_slot)
```

```
            else if replica.wedged_history is Empty then
                    history_difference = (longest_history – replica.checkpoint_proof.last_slot)
            endif
            send ('catchup_message',  (SHA256(history_difference), encrypt(history_difference,
                                        olympus.private_key))) to replica
    olympus.timer_for_caughtup = start timer();
    if  olympus.timer_for_caughtup expires then
            // suppose a replica dies. We cant wait forever until the replica comes back
            olympus.invalidate_and_select()
```

---

handle_caughtup_message: This method is responsible for checking if all the replicas have the same hash of their running_state and then picks a random replica, to which, it sends get_running_state message.
There is a "timer_running_state" which starts ticking after sending the 'get_running_state' message. On expiry, it calls **invalidate_and_select** method
This method is triggered when replicas send their running state hash after doing the operations defined in the catchup request.

```
olympus.handle_caughtup_message(replica, caughtup_message)
    extract { hashed_state } from decrypt(caughtup_message, replica_public_key)
    if caughtup_message cannot be decrypted then
            olympus.invalidate_and_select()
    endif
    olympus.caughtup_response =  olympus.caughtup_response U hashed_state

    else if len( olympus.caughtup_response) == len( olympus.set_of_quorum ) then
            if all olympus.caughtup_response have the same hashed_state then
                    random_replica = select a random replica from olympus.set_of_quorum
                    olympus.hashed_state_for_later_reference = hashed_state
                    olympus.timer_running_state = start timer()  //start timer for getting running state
                    if olympus.timer_running_state expires then
                            olympus.invalidate_and_select()
                    endif
                    send ('get_running_state', (nonce, sign (nonce) with olympus.private_key)) to
                                                            random_replica_id
            else if there is some inconsistent hash of the state then
                    olympus.invalidate_and_select()
            endif
    endif
```

---

invalidate_and_select: This method invalidates its quorum and starts over the entire process of selecting new quorums, finding longest history and sending catchup requests to the quorum.

```
olympus.invalidate_and_select()
    invalidate olympus.set_of_quorums
```

```
        invalidate olympus.hashed_state_for_later_reference
        olympus.select_quorum()
        olympus.find_longest_history()
```

---

handle_running_state: This method is triggered when it receives a running state message from a replica.
It first, tries to decrypt the signed request to validate if the desired replica has replied.
Then it compares the hash of the running state with its own reference hash which it computed earlier(**hashed_state_for_later_reference**).
If they match, then Olympus “creates new replicas” by seeding its initial state and history as NULL. It also stops the **timer_running_state**.
If they don't match, then a Olympus tries to select a new set of quorums and the process repeats.

```
olympus.handle_running_state(replica, message)
        extract { signed_request } from message
        running_state = decrypt(signed_request, public key of replica)
        if signed_request is decrypted then:
                stop olympus.timer_running_state
                if SHA256(running_state) == olympus.hashed_state_for_later_reference then
                        olympus.create_new_replicas(initial_state = running_state, history=NULL)
                else
                        olympus.invalidate_and_select()
                endif
        else
                // sender is not authentic:
                drop the request and return
        endif
```

---

handle_proof_of_misbehavior: This method is used for validating misbehavior in the system. The handler can be triggered by both client and replica.
Client needs to provide result_proof as evidence, whereas, Replicas can show either result_proof or order_proof
If misbehavior is validated, then Olympus decides to reconfigure.

```
olympus.handle_proof_of_misbehavior(sender, message)
        if sender is client then do
                extract { result_proof } from message
                return olympus.validate_result_proof_misbehavior(result_proof)
        else if sender is replica do
                if message contains 'order_proof' then do
                        extract { order_proof } from message
                        return olympus.validate_order_proof_misbehavior(order_proof)
                else do
```

```
                extract { result_proof } from message
                return olympus.validate_result_proof_misbehavior(result_proof)
        endif
endif
```

---

validate_result_proof_misbehavior: This method is for validating the misbehavior of result_proof It check if all hashes in the result_proof are same

---

olympus.validate_result_proof_misbehavior(result_proof)

```
extract { result, complete_result_proof } from result_proof
hashed_result = SHA256(result)
count = 0
for each (index, proof) in enumerate(complete_result_proof):
        if hashed_result == decrypt(proof, public key of replica[index]) then
                increment count
        endif
if count > t+1 then
        olympus.reconfigure()
else
        ignore request
endif
```

---

validate_order_proof_misbehavio: This method validates the misbehavior of the order_proof. It checks if same slots have conflicting operations.

---

olympus.validate_order_proof_misbehavior(order_proof)

```
extract {  slot, desired_operation, order_proof_1 } from order_proof
check if for that slot, there are operations other than desired_operation in the order_proof_1:
if true:
        olympus.reconfigure()
else:
        ignore request
endif
```
