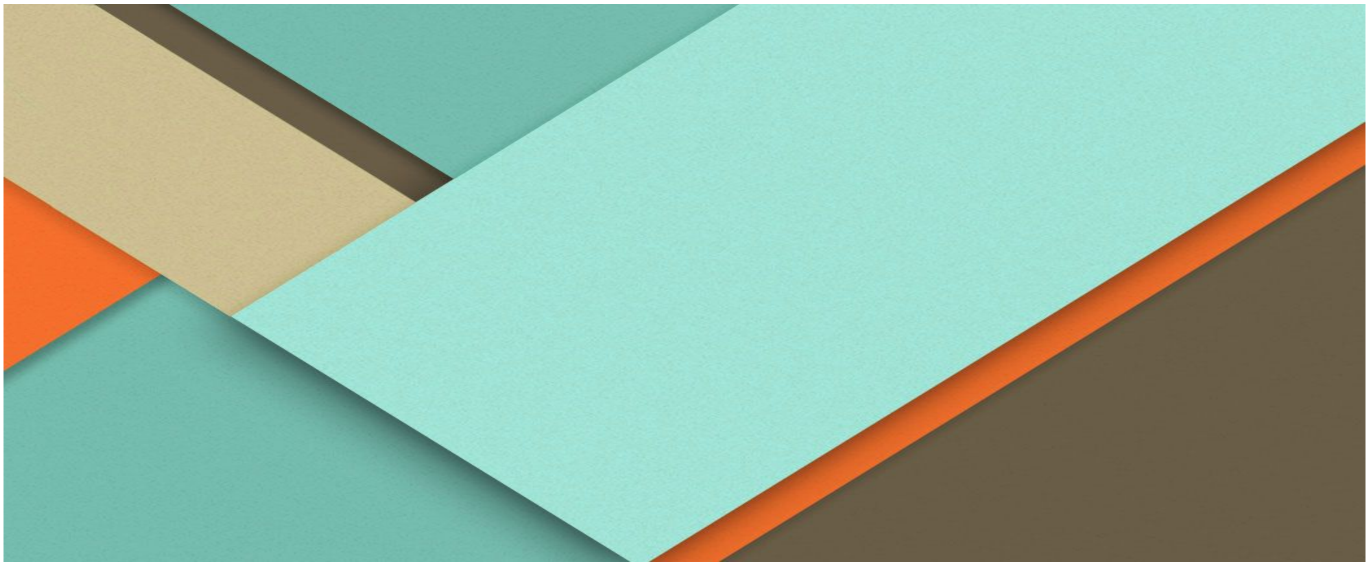

Byzantine Chain Replication Pseudocode

COURSE : CSE 535, ASYNCHRONOUS SYSTEMS, Prof. Scott D. Stoller

DATE: 09/22/2017

Contributors:

MRIDUL RANJAN (NetId: mranjan, SBU ID: 111482003)

ROHIT KUMAR (NetId: rokkumar, SBU ID: 111471168)

Overview

This document includes the pseudo-code for clients, olympus, and replicas using public key cryptography for signature to build an asynchronous system that is byzantine fault tolerant. It uses the strategy as described in the paper published by Robbert van Renesse, Chi ho, and Nicolas Schiper on **"Byzantine Chain Replication, Dec 2012".**

1. CLIENT:

###

#1. Send a query to Olympus to fetch the configuration details of the replicas.

#2. Defines a handler for the response received from Olympus for the configuration query. The response will have the following:

a. Sequence of replicas in the chain. The first in the sequence represents the Head of the chain and next subsequent are the replicas in their order of arrival in the chain. For Ex: Head ⇔ Replica 1 ⇔ Replica 2 ⇔ Replica 3 ⇔ Tail, will be represented in the ordered sequence {Head, Replica 1, Replica 2, Replica 3, Tail}

b. All the public keys of all the replicas. This will be used for verifying the signed result proofs received from the tail. This will be in the same order as the sequence of replicas. So, for the above chain the public keys will be in the following order - {PKey(Head), PKey(Replica 1), PKey(Replica 2), PKey(Replica 3), PKey(Tail)}

c. Current configuration Id for the sequence of replicas. This will be used to determine if the current configuration has changed or not.

#3. Sends request to olympus for reconfiguration by providing proof of misbehaviour. The message format for this will be:

<msg type, Client Id, Message Identifier, <Result proof>, <Proof of misbehaviour>>

- *msg type:* Type of request.
- *Client ID:* Must uniquely determine the client
- *Message Identifier:* Must uniquely determine the order.
- *Result Proof:* The result proof received from any replica as response to the order request.

- *Proof of misbehaviour:* Conflicting result statements from any of the replica in the result proof.

NOTE: Signature mismatch cannot be considered as a proof of misbehaviour as an intruder can impersonate a replica with wrong signature.

#4. Sends a request order to the head of the chain. It must also trigger a timer for the current request. The message format will be: *<Msg type, Client Id, Message Identifier, Operation>*

#5. Sends a retransmit message for which timeout happened. The message format for this will be: *<Msg Type, Client Id, Message Identifier, Operation>*

NOTE: Client must reuse the original message identifier.

```
#################################################################
#########################
## PSEUDO CODE FOR CLIENT ##
#########################
------------------------------------------------------------------------------------------------------------------------------

# Global Configurations
replicas=[]
current_configuration_id = “”
public _key_map = {}

def handler_config_respose(response):
        replicas =  response.replicas
        current_configuration_Id =  response.configuration_id
        public_key_map =  response.public_key_map
```

```
def handler_order_respose(response):
    if(verify_response(response)):
            if(proof_of_misbehaviour_present()) :
               return True
    return False

def execute_order():
   was_request_served=True
   retry_all = False
   ### while ends here ###
   while(True):
       config = send_configuration_query_to_olympus('GET_CONFIG', client_id, msg_id)
       if (config is not set)
            wait()
            response = receive('CONFIG_RESPONSE', msg_id)
            handler_config_respose(response)
            continue  #if ends here
       if(was_request_served):
            request = get_next_request_from_application(client_id)
            operation = get_next_operation()  #if ends here
       timer = start_timer(<timeout_value>)
       msg_id = get_unique_msg_id()
       head_replica_id = get_head();
       if( not(retry_all)):
            send_order('REQUEST_ORDER', client_id, head_replica_id, msg_id, operation)
            wait();
            response = received('ORDER_RESPONSE', msg_id, result_proof);
        else:
            for r in replicas:
```

```
            # this is the retransmit message to all the replicas
            send_order('REQUEST_ORDER', client_id, r.replica_id, msg_id, operation)
        wait()
        response = received('ORDER_RESPONSE', msg_id, result_proof);
    # if - else block ends here
    if( response is present and timer not expired)
        stopTimer()
        was_request_served=True
        retry_all = False
        if(handler_order_respose(response)) :
            send_reconfigure_request(client_id, msg_id, result_proof,    \
              proof_of_misbehaviour)
    if (timer expired) :
        was_request_served=False
        retry_all = True
### while block ends here ###
### execute order ends here ###

def main():
    client_id = generate_unique_client_id()
    execute_order()
```

Function Explanations:

- execute_order() - Executes the order for the client.
- get_next_request_from_application() - Fetches the next request from the application
- send_configuration_query_to_olympus() - Fetches the current configuration from Olympus.
- send_reconfigure_request() - Send a request to reconfigure the configuration with proof of misbehaviour.

- send_order() - Send an order to head of the chain.
- start_timer(<timeout>) - starts a timer for the specified timeout.
- get_unique_client_id() - Returns the client Id.
- get_next_operation() - Returns the next operation for the client.
- handler_order_resposne() - Parses the response received for an order. Validates if misbehaviour is present or not.
- handler_config_resposne() - Parses the response received for the configuration. Stores them in the client as a global configuration.

----

2. OLYMPUS:

##

#1. Initializes the configuration by reading the configuration file.

a. Olympus must be able to read config file for initializing parameters -

- number of failure tolerant replicas (t): maximum number of replicas that may become faulty in the chain. Based on this, the Olympus will configure (2*t + 1) replicas and (t+1) quorums.
- global timeout for each replica: maximum timeout for which a replica may wait for result proof.
- number of result proof to cache - maximum number of result proofs that can be cached at each replica.
- checkpoint threshold: maximum number of order requests that are persisted. An order is persisted if the head received a correct result proof for that order.

#2. Handle the reconfiguration request from client after validating proof of misbehaviour

#3. Handle the reconfiguration request from the replicas.

#4. Respond to send_configuration_query_to_olympus() from client.

```
#################################################################

  ###########################
  ## PSEUDO CODE FOR OLYMPUS##
  ###########################
------------------------------------------------------------------------------------------------------------------------
def initialize_configuration(<conf_file>):
    #Read configuration from the <conf_file> file
    replicas = create_replicas((2 * num_of_faulty_replica) + 1)
    set_quorum(num_of_faulty_replica + 1)
    public_private_keys = create_public_private_keys(<list of replicas>)
    head = set_head(replicas[0])
    tail = set_tail(replicas[2*t])
    init_hist={}
    map_of_public_key_for_all_replica = get_public_keys(replicas)
    for r in replicas:
        send_msg_to_replica(r.replica_id, getPrivateKey(r), head, tail, init_hist,
            checkpoint_threshold, global_timeout, map_of_public_key_for_all_replica)
   send_all_replica(ACTIVE)

def reconfigure_system(history):
    send_all_replica(IMMUTABLE)
    initialize_configuration(<conf_file>)
    send_new_history_to_all_replica(history)
    send_all_replica(ACTIVE)

def construct_new_history_from_wedge():
    while(response is not  having consistent_valid_history from all quorums):
        issue_wedge_statement()  # send to all replicas
```

```
          response = wait_for_response_from_all_replicas_in_quorum()

          ## This is slot-operation sequence across histories of all wedged statements
          longest_history = get_longest_sequence(response)
          for (replica in response.quorum_replicas):
              do_catch_up(longest_history - replica.history)
          caught_up_responses = get_caught_up_response_from_all_replicas in_quorum()
          if (verify_hash(caught_up_responses)):
              state_hash = get_verified_hash_value(caught_up_responses)
              current_state = get_current_state_from_some_replica_in_quorum()
              ## This method will converge as quorum will have some correct replica
              while (hash(current_state) != state_hash):
                 current_state = get_current_state_from_some_replica_in_quorum()
                 return current_state
          else:
             ## quorum has some faulty replica which has deviated after checkpointing
             continue

def serve_request(request):
    if(request.type == CONFIG_QUERY):
         return fetch_current_config()
    if(request.type == RECONFIGURE_REQUEST and request.Id equals client.id ):
         if( not (validate_proof_of_misbehaviour( request.proof_of_misbehaviour))):
                # Client seems to be Byzantine. So either we ignore or block the client.
                ignore_or_block_client(request.client_id)
                 return
    if(request.type == RECONFIGURE_REQUEST):
            # Issue a wedge statement to all replicas
           # Await Response till quorum
```

```
            # Calculate new history
             history = construct_new_history_from_wedge()
             reconfigure(history)

def main():
    initialize_configuration(<conf_file>)
    while(true):
        request = wait_for_any_request()
        if(request is received):
            serve_request(request)
```

----

3. Replica:

```
###################################################################
 #1. Handler for configuration message from Olympus, once replica is instantiated
     successfully. The replica applies all the configuration parameters provided by
     the Olympus in the first message. The first message received from Olympus
     contains the following:
         -  Replica Id: Uniquely determines the replica
         -  PrivateKey: private key for self.
         - Head Replica Id:  the replica Id for the head.
         - Tail Replica Id:  the replica Id for the tail.
         - Initial History: the starting history for the replica.
         - checkpoint threshold:  Number of order requests that are persisted.
           An order is persisted if the head received a correct result proof for that order.
         -  global timeout: maximum timeout for which a replica may wait for result proof.
        -  number of result proof to cache - maximum number of result proofs that can be
            cached at each replica.
```

- map_of_public_key_for_all_replica: map of the replica Id to public key for all replica

#2. Handle message to change the current mode of the replica from Olympus

#3. Handle message to update the history of the replica from Olympus

#4. Handle a wedge request from Olympus

#5. Head must be capable to initiate checkpoint proof.

#6. Handle a checkpoint proof

#7. Head must create a shuttle with order proof and result proof for requested order from client

#8. Handler for shuttle received from preceded replica. Tail should do additional work of sending result proof to client as well as sending the completed proofs to head across chain in reverse order.

#9. Handle retransmitted message from client

- Send error code to client if the current mode is IMMUTABLE
- Respond client with cached result proof
- Forward the request to head and start a timer. If the timeout happens trigger reconfiguration message to Olympus.

#10. Handler for shuttle received from successor replica

#11. Verifying the ordered proof in the shuttle for misbehaviour. Trigger a reconfiguration message to Olympus

#12. Handler for exposing current running state to Olympus.

#13. Handler for catch_up request from Olympus

NOTE: Following are the ways in which a replica would find misbehaviour of other replica :

1.) validating order proofs from predecessor replicas for conflicting slots.

2.) validating result proofs when they are returned from tail after completion.

##

```
##########################
## PSEUDO CODE FOR REPLICA##
##########################
```

```
-----------------------------------------------------------------------------------------------------------------------
# This sets all the parameters from Olympus on initialization and reconfiguration
def handler_configuartion_message_from_olympus(r.replica_id, private_key, head, tail,
        init_hist, checkpoint_threshold, global_timeout, map_of_public_key_for_all_replica)):

# This sets the mode of replica - ACTIVE, PENDING, IMMUTABLE
def handle_message_to_change_current_mode_by_olympus(mode):

def handle_message_to_update_history_from_olympus(history):
       # Request Type = UPDATE_HISTORY_REQUEST
    if(mode == 'PENDING')
        self.hist = history
    else :
        # History cannot be reset when replica is active or immutable
        error

def handle_wedge_request_from_olympus():
    # Request Type = WEDGED_REQUEST
   if (mode == 'ACTIVE')
       mode = 'IMMUTABLE
        return history, latest_checkpoint
```

```
def initiate_checkpoint_proof_from_head():
     # Request Type = INIT_CHECKPOINT_REQUEST
    if (replica_id == head):
        if (get_current_state_count() >= checkpoint_threshold):
            adds <checkpoint, hash(current_state)-signed-by-head> to check-point proof
            forwards_shuttle_to_next_replica(check-point-shuttle)

def handle_checkpoint_proof(check-point_shuttle):
        # As pointed in the discussion, it's safer to include validation but not necessary
        validate(checkpoint-proof)
        receive_from = checkpoint_shuttle_received_from(checkpoint-proof)
        switch(receive_from):
            case predecessor:
                add<checkpoint, hash(current_state) - signed by self > to checkpoint-proof
                if (replica_id != tail):
                    forward_shuttle_to_next_replica()
                else:
                    reverse_shuttle_predecessor()
            case successor:
                 remove corresponding prefix from current state
                 Update_latest_checkpoint
                 reverse_shuttle_predecessor()

def handle_request_from_client(command):
    if(replica_id == head):
        current_slot++
        Create_signed_order_statement <order, slot, command>
        apply command to its running state and obtain result r
        create_signed_result_statement<result, r>
        Create shuttle and add order_proof and result_proof to it
        forward_shuttle_to_next_replica()

def handle_shuttle_from_preceding_replica(shuttle):
    if (is_valid_order_proof(shuttle.order_proof)):
        Create_signed_order_statement <order, slot, command>
        apply command to its running state and obtain result r
        add order_statement to shuttle.order_proof
        add result r to shuttle.result_proof
        if (replica_id != tail):
            forward_shuttle_to_next_replica()
        else:
            send_result_proof_to_client()
            cache_result_order()
            send_shuttle_to_predecessor_reverse_on_chain()
    else:
        request_olympus_for_reconfiguration()

# if all preceding replica have signed order_proof with same slot and operations
# and there is no conflict for slot, then return true else false
def is_valid_order_proof(order_proof):

def handle_retransmitted_request_from_client(request):
    if (request.message_id is cached in results):
        return get_cached_result(request.message_id)
    if (request_id != head):
        timer = start_timer()
        send_request_to_head(request)
        response = get_response_from_head(request)
```

```
          if (time_out):
              request_olympus_forreconfiguration()
          else:
              cancel_timer(timer)
              return response
     else:
         if (order_proof contains request.operation
                     but reverse_result_proof_not_received_at_head):
         ## simply start timer and if result_proof not received till time_out then request
         ##   reconfiguration
         timer = start_timer
         result_proof = get_reverse _result_proof_from_succsesor_node(message_id)
         if (time_out):
              request_reconfiguaration_from_olympus()
         else:
              suspend_timer(timer)
              return result_proof

# reverse shuttle has completed order proof and result proof. Here order proofs are
# committed to local history and result proofs are cached after validation. If proof of
# misbehaviour is found, then request for reconfiguration is raised.
def handle_shuttle_received_from_succesor():

def handle_catch_up_request_from_olympus(<sequence of statements> orders_stmts):
     apply_order_statements_in_sequence(order_stmts)
     update_the_current_resulting_running_state()
     return hash(current_resulting_state)
```

```
def serve_request(request):
    if(request.type == CONFIGURATION_REQUEST_FROM_OLYMPUS):
        return handler_configuartion_message_from_olympus()
    if(request.type == CHANGE_CURRENT_MODE):
        return handle_message_to_change_current_mode_by_olympus(request.get_mode())
    if(request.type == UPDATE_HISTORY):
        return handle_message_to_update_history_from_olympus(request.get_history())
    if(request.type == WEDGE):
        return handle_wedge_request_from_olympus()
    if (request.type == HANDLE_CHECKPOINT_PROOF ):
        return handle_checkpoint_proof(request.get_checkpoint_shuttle())
    if(request.type == COMMAND):
        return handle_request_from_client(request.get_command())
    if(request.type == FORWARDING_SHUTTLE):
        return handle_shuttle_from_preceding_replica(request.get_shuttle())
    if(request.type == COMMAND_RETRNASMITTED):
        return handle_retransmitted_request_from_client(request)
    if(request.type == REVERSE_SHUTTLE):
        return handle_shuttle_received_from_succesor(request.get_shuttle())
    if(request.type == CATCH_UP):
        return handle_catch_up_request_from_olympus(request.get_ordered_sequence())


def main():
    if (replica_id == head):
        ## After fixed checkpoint threshold number of requests received from client,
        ## head should start checkpoint proof and trigger it down the chain
        initiate_checkpoint_proof_from_head()
    while(true):
        request = wait_for_any_request()
```

```
if(request is received):
      serve_request(request)
```

--- *END* ---