CSE 535: Distributed Systems

Team- Name: Evil Geniuses

1) Nihal Goalla (113276929)
2) Rohith Vaddavalli (113261811)
3) Manoj Kumar Ravuri (113262634)

In this file, we are just adding the extensions to the pseudocode mentioned in the DiemBFT v4 paper.

Ledger Module

pending_map // map of all the pending blocks with block_id as key

commited_blocks:// map of all committed block to prevent deduplication, and to get recently committed blocks

commit_state_id // state id of the ledger which is defined as commit_state_id = hash(parent_commit_state_id+str(txns))

```
Function Ledger_speculate(prev_block_id, block_id, block):
    pending_map[block_id] <- (“prev_block_id” <- prev_block_id, “block” <- block )


Function Ledger_pending_state(block_id):
    If block_id in pending_map  then
         txn <- pending_map[block_id][“block”].payload
        return  hash( commit_state_id + txn)
    else
        return ⊥

Function Ledger_commit(block_id):
    If block_id in pending_map  then
        block <- pending_map[block_id][“block”]
         txn <- block.payload
        If  ‘dummy_txn’ not in txn then
            write(txn)
            send Message(‘committed ‘ +txn,request_map[txn]) // this sends an acknowledgement to the requested client
    committed_blocks[block_id]=pending_map[block_id]
    pending_map.pop(block_id)
```

```
Function Ledger_comitted_block(block_id):
    If block_id in committed_blocks  then
        return committed_blocks[block_id][“block”]
    return $\perp$
```

Mempool Module

processed_transactions // to keep track of processed transactions, basically acts as a cache of processed transactions

mem_queue // keep track of what transactions are yet to be processed

```
Function MemPool_get_transactions(proposed_txns):
    If proposed_txns then
        processed_transactions <- processed_transactions $\cup$ proposed_txns
    If |mem_queue|=0 then
        await(|mem_queue|)
    If proposed_txns and proposed_txns in mem_queue then
        mem_queue.remove(proposed_txns)
    else
        new_txn=mem_queue.pop()
        If new_txn in processed_transactions then
            return  MemPool_get_transactions($\perp$)
        processed_transactions <- processed_transactions $\cup$ new_txn
        return new_txn
    return $\perp$
```

```
Function add_to_queue(M):
    mem_queue.insert(M)
```

Signature verification

id://used to distinguish between clients based on their ids
message:// message is encrypted by the client or replica by their private key
type: // check if the signature is sent by replica or client
replica_process: // it is used to find the replica from the signature in leader election

//To make the signature from the block

```
Function Safety_make_signature(block):
    return Signature(replica_id, private_key.sign(block.payload.encode('utf-8')), 'replica', curr_pr)
```

//To verify the signature sent by the client or replica

```
Function Safety_verify_signature(id, message, type='replica'):
    if type == "replica":
        v_key = VerifyKey(self.replica_public_keys[id],
                          encoder=HexEncoder)
    elif type == 'client':
        v_key = VerifyKey(self.client_public_keys[id],
                          encoder=HexEncoder)
    try:
        v_key.verify(message)
    except BadSignatureError:
         return False
    except:
        return False
    return True
```

//To verify all the signatures

```
Function Safety_valid_signatures(high_qc, last_tc):
    i = 0
    if (high_qc.vote_info.round == -1):
        return True
    for signature in high_qc.signatures:
        if Safety_verify_signature(signature.id, signature.message, signature.type):
            i += 1
    if (i == 2*f+1):
        return True
    else:  return False
```

// To sign the command sent by the client

```
Function sign_cmd(cmd):
    if not faulty_client:
        return Signature(self.client_id, self.private_key.sign(cmd.encode('utf-8')), 'client',None)
    else:
        faulty_client=False
        faulty_private_key=SigningKey.generate()
        faulty_public_key=faulty_private_key.verify_key
        return Signature(client_id, faulty_private_key.sign(cmd.encode('utf-8')), 'client', None)
```

Client Logic

cmds_pending //tracks requests which have not yet sent acknowledgement to client
f // number of faulty nodes
rep_keys // set of replica keys to authenticate acks

```
client_key // private key of client
client_id // client’s unique id
replica_timeout//
number_of_requests// number of requests to send to clients
request_gap// request gap between each request
requested_root// requested root to send empty transitions to commit last blocks
terminate// stops terminating till it is set to True


Function broadcast_request(command,replicas):
    signature=sign(cmd)
    send( (‘request’,cmd, logical_clock(),client_id,signature ) ,to =replicas)
    If await( each( id in rep_keys, has= received(‘request_ack’,id’) then pass
    else if timeout(replica_timeout) : broadcast_request(cmd,reps)

Function receive_committed(msg=(‘committed’,cmd,replica_id) ):
    cnt = 0
    If ‘dummy_txn’ not in cmd then
        cmds_pending[cmd]  <- cmds_pending[cmd]+  1
        for  cmd_ in cmds_pending:
            If cmds_pending[cmd_] >= 2*f +1 then
                cnt = cnt + 1
    If cnt==(number_of_requests-2) and not requested_root then
        send( ‘ send dummy txns’,logical_clock(),to=parent() )
        requested_root = True
    else if cnt==number_of_requests then
        terminate = True

Function run():
    for i in range(number_of_requests):
        sleep(number_of_requests)
        cmd=str(client_id)+”---->”+str(i)
        broadcast_request(cmd,replicas)

    await(terminate)
    send( ('client_done',client_id,logical_clock() ) ,to=parent() )
```

Replica Logic

```
replica_id // unique replica_id
f           // number of faulty nodes
replicas // all replicas
```

```
processing_req //  False
replica_map // rep_map
inv_map // map replica processes to ids
client_map // maps client ids to client processes
request_map // map requests to clients
terminate // stops terminating till it is set to True

Function receive_proposal(msg=('proposal', p1, c1,sid)):
        Main_start_event_processing(p1, 'proposal_message')

Function receive_vote(msg=('vote', vote_info, c1,rid)):
        Main_start_event_processing(vote_info, 'vote_message')

Function receive_client_request(msg=('request', cmd, c, p,signature)):
        If Safety_verify_signature(signature.id, signature.message, signature.type)
                M = ('request', cmd, c)
        If cmd in request_map then
          send( ('request_ack',replica_id),to=client_map[p])
        //Above check is to prevent processing a duplicate request from client
        else::
          request_map[cmd]=client_map[p]
          send( ('request_ack',replica_id),to=client_map[p])
          Main_add_to_Mempool(M)

Function receive_root_request(msg=('request', cmd, c, p)):
     If (p==parent() ) then
        M = ('request', cmd, c)
        send(('request_ack',replica_id),to=parent())
        Main_add_to_Mempool(M)

Function receive_timeout_msg(msg=('time_out_msg', timeout_msg,c)):
     Main_start_event_processing(timeout_msg,'timeout_message')

Function receive_terminate_signal(msg=('done',sender)):
     if sender==parent():
        send(('replica_done',replica_id),to=parent())
        PaceMaker_stop_timer(current_round)
        terminate=True


Function run():
        sleep(5)
     If (Main_can_send()) then
        Main_process_new_round_event()
```

```
While not (self.terminate):  \\This keeps the replica running till termination
    If current process is the leader:
            yield for messages
```

Sync-up replicas

The leader along with the transaction will also send the latest_index in its own DB.

If any of the followers has the latest_index which has more than 1 difference than that of the leader (diff(leader_latest_db_index, follower_latest_db_index) > 1); it means one of the nodes is out of sync.

When a replica figures out it is out-of-sync, it does the following function.

We are going to use a similar system as in RAFT where we will index all the commits in the ledger.

When a replica is out of sync:

We will fetch the indices of the latest commits in all the neighboring replicas.

Based on the replies we get, we will form a consensus on what the correct latest index might be.

When we form a consensus on the latest index, the out-of-sync replica will try to fetch the data until the latest index and appends to its own ledger.

In this way, we can sync out-of-sync replicas.

We can’t let the leader force its own log on to its followers (which is what happens in RAFT), because we might not know how truthful the leader is.