1 Scenario Generator

Modifications done to submitted Phase 3 pseudocode

- Changes to the implementation of generate_heuristic_partitions method.

- Deterministic partition generation algorithm is a backtracking based algorithm which generates all possible n nodes into k subsets.

- The above algorithm generates Stirling number of second type number of partitions.

- Filter is applied to filter out partition-sets without a single quorum-partition(atleast one partition which has quorum(without including twin)).

- For randomized partition generation algorithm, we first shuffle the nodes list and generate deterministically. The generated list is again randomly shuffled simulating a total random generation experience.

- Made changes to is_valid scenario method which is used to potentially prune non-responsive cases.

```


iterator  // Global iterator in case of Deterministic Enumeration Order

Procedure scenario_generator(nodes, twins, n_partitions, n_rounds,
    partition_limit, partition_leader_limit, max_testcases, random_seed, is_Faulty_Leader,
    is_Deterministic, file_path):

    partitions_list = generate_hueristic_partitions(n_partitions, nodes, twins, partition_limit,
      is_Deterministic, seed)

    partition_leader_list = []
    partition_leader_list_high_prob = [] //Fill this list with quorum partitions with high probability.

    partition_leader_list_low_prob = [] //Fill this list with quorum partitions with low probability.

    for partition from partitions_list:
        if is_Faulty_Leader:
            nodes = [i if isFaulty(i) for i in nodes] // Filtering only faulty nodes if
            is_Faulty_Leader is set.
        for node in nodes:
            if(len(partition_leader_list) == partition_leader_limit):
                break
            partition_leader_list_high_prob.add(node : quorum_partition)
            partition_leader_list_low_prob.add(node : non_quorum_partition)
            partition_leader_list.add(node : partition)

    num_scenarios=0
    while num_scenarios < max_testcases:
        scenario = create_scenario(partition_leader_dict, random_seed, is_Deterministic, n_rounds)
        if is_valid(scenario):
            file.flush(scenario, file_path)
        num_scenarios++

Function is_valid(scenario):
    // This check is used to prune potentially non-responsive cases.

    Let l_r, l_r1, l_r2 are leaders of r, r+1, r+2nd rounds respectively.

    counter c=0

    Check if l_r, l_r1 are in quorum partitions in round r
    and l_r1, l_r2 are in quorum partitions in round r+1
    and l_r2 is in quorum partitions in round r+2
    and increment counter c

    return True if c>threshold else return False


Failure_Type {
    WILDCARD : 1,
    PROPOSE : 2,
    VOTE : 3
}

Function get_tuple(is_Deterministic, partition_leader_dict, failure_type):

    if is_Deterministic:
        item = partition_leader_dict[iterator++] // Get item at location iterator
        deterministically and increment iterator
    else :
        id = floor(random.uniform(0, 1)*len(partition_leader_dict))  // Get item randomly
        from the partition_leader_dict
        item = partition_leader_dict[id]

    return new Tuple<item.key, item.value, failure_type>


Function create_scenario(partition_leader_dict, seed, is_Deterministic, rounds):
    random.seed(seed)
    scenario={}

    // For each round, determine if a failures are being introduced. Accordingly make
    amends to partition-leader combination and append it to scenario.

    for round from rounds:
        introduce_failure = random.uniform(0, 1)
        if introduce_failure < 0.8:
            scenario.put([{round, get_tuple(is_Deterministic, partition_leader_dict,
            Failure_Type.None)}])

        else if introduce_failure < 0.9:
            tuple1 = get_tuple(is_Deterministic, partition_leader_dict, Failure_Type.None)
            tuple2 = get_tuple(is_Deterministic, partition_leader_dict, Failure_Type.PROPOSAL)

            make_singleton(tuple1.partition, tuple1.leader) // Make all partition with leader-
            singleton set(Just the leader). This is to replicate intra-partition drop
            make_singleton(tuple2.partition, tuple2.leader)
            scenario.put([ {round, tuple1}, {round, tuple2} ])

        else:
            tuple1 = get_tuple(is_Deterministic, partition_leader_dict, Failure_Type.None)
            tuple2 = get_tuple(is_Deterministic, partition_leader_dict, Failure_Type.VOTE)

            make_singleton(tuple1.partition, tuple1.leader)
            make_singleton(tuple2.partition, tuple2.leader)
            scenario.put([ {round, tuple1}, {round, tuple2} ])


Function generate_heuristic_partitions(num_partitions, nodes, twins, partition_limit, is_Deterministic
, seed):

    global partitions
    f = len(twins)
    n = len(nodes)
    total_nodes = nodes + twins

    if is_Deterministic:
       deterministic_partition_gen_algorithm(0, nodes, k, 0, results, partition_limit)
        return partitions

    else:
        random.shuffle(total_nodes)
        deterministic_partition_gen_algorithm(0, nodes, k, 0, results, partition_limit)
        random.shuffle(partitions)

Function deterministic_gen_algorithm(i, nodes, k, nums, results, partition_limit):
    i iterates over all nodes and nodes[i] is positioned into all the possible subsets until we
    encounter the first empty subset. Positioning nodes[i] into a subset results in a recursive call.
    results is an intermediate list and is appended to partitions
    if k partitions are filled using all the nodes.

    if i >= len(nodes):
        if nums == k: Used to check if we populated all k partitions of not.
            partitions.append(results)
        return

    for j in range(len(results)):
        add nodes[i] to results[j] subset.
        if length of results subset is greater than 1:
            deterministic_gen_algorithm(i + 1, nodes, k, nums, results, partition_limit)
            pop last element from results
        else:
            deterministic_gen_algorithm(i + 1, nodes, k, nums + 1, results, partition_limit)
            pop last element from results
            break
```

---

2 Scenario Executor

Modifications done to submitted Phase 3 pseudocode

- Added Safety Check to the Network Playground. The ledgers are checked after each commit to check ordering in all non faulty validators. Process stops with Safety Violation when this violation happens
- Termination happens on one of the following cases: Liveliness Violation , Safety Violation , after Successfull commit of all commands to ledger, all process crossed the max rounds configured for the test case
- Other changes are minor and only pertaining to implementation and not the design.

Sync up Replicas that got behind

- A replica realises that it is behind on receiving a proposal block with a qc that was formed in a round much ahead of its current state.
- A sync up request with the behind replica's high_qc is sent back in all these cases
- Upon receiving this sync up request the validators replies with a list of QC's that are missing after the requester's high qc in the root to leaf path of its block tree.
- The behind replica processes these QC's and updates itself after verifying the signatures on the QC.

Mempool Issue

- So Far, Txns in mempool is removed on proposing block or on receiving propose
- But not all proposed blocks are guaranteed for a commit as it might be pruned when appropriate qc's are not formed
- Hence this logic is Changed, mempool txns are removed only after commit. Propsed or Received Txns are cached so that everytime we first propose the commands that are not proposed yet.

NetworkPlayground.da

```
self.last_executed_round={}
self.txn_commit_order = For Safety Check
self.last_commited_round = {}
self.msgcount_per_round={}
self.nocommit_pool = []
//Note: All config variables are initialized during setup. Accessed by config.

//Returns partition based on scenario and intended destination of msg
function get_partition_and_destination(source, round, msgtype):
    partition = []
    intended_destination = []
    PLF_list = config.scenario[round] //(Part'n change handled as per round
    for ipartition, failtype in PLC_list: //(Partition, Leader, FailType) list
        if source in ipartition and (!failtype or failtype == msgtype):
            partition = ipartition //Failtype: for intra part msgdrop if configured
    if msgtype == PROPOSE or msgtype == TIMEOUT:
        //Intended Destination: All Processes (Validators and Twins)
        intended_destination = config.validators + config.twinValidators
    if msgtype == VOTEMSG:
        //Intended Destination: Next Round Leader(and twin if available)
        next_leader = config.round_leader[round+1]
        intended_destination = next_leader
        if next_leader in config.twin:
            intended_destination = intended_destination + config.twin[next_leader]
    return partition, intended_destination

//Sends/Drops messages to destinations as per the test case
function handleMsg(source,round, msg, msgtype):
    partition, intended_destination = get_partition_and_destination(source, round, msgtype)
    destination = []
    for validator in intended_destination:
        if validator in partition:
            //Intersection btwn Intended Destination and Partition
            destination = destination + {validator}
     //Msg Redirect as per Configs
    if source in config.twin:
        //DistAlgo Specific: Send visible process id before Redirect'n if twin
        send((msgtype, msg, config.twin[source]), to=destination)
    else:
        send((msgtype, msg, source), to=destination)

//Checks for Liveness Violation when only TCs are Formed
//Violated if 2f+1 Nodes did not commit for a Threshold Number of Rounds at same time
//Threshold is selected based on max_round possible,nclient instructions
function check_nocommit_pool(source):
    if self.last_executed_round[source] - self.last_commited_round[source]
                                                    >= config.LIVE_THRESHOLD:
```

```
        self.nocommit_pool.add(source)
    if |self.nocommit_pool| == 2f+1: // worst case: f+f faulty (including twins)
        send((’Done’,’LIVENESS_VIOLATION’), to=main)

//Remove From No Commit Pool if Possible on Commit
function update_nocommit_pool(source):
    if source not in nocommit_pool:
        return
    if self.last_executed_round[source] -
    self.last_commited_round[source] < config.LIVE_THRESHOLD:
        self.nocommit_pool.remove(source)

//Check if majority of validators reached max_round in the execution
//Remaining validators cannot progress by qc as there wont be any quorum
function check_process_completion():
    ncompleted = 0
    for validator in {validators U twinValidators}:
        if self.msgcount_per_round[source] > 10 or len(txn_commit_order[source]) == config[’nops]:
            ncompleted = ncompleted + 1
    if ncompleted == 2f+1:
        //if 2f+1 ( twins included) had completed, then no possibility of sync up
        send((’Done’,’Commits Successfull’), to=main)

function safety_check(source, txns):
    last_commited_index = txn_commit_order[source] - 1
    out_of_order = False
    for command in txns:
            txn_commit_order[source].append(command)
            last_commited_index = last_commited_index + 1
            for i in range(0, len(validators)):
                if i in config[’twin’]:Not Needed for faulty Nodes
                    continue
                icommit_order = txn_commit_order.get(i, [])
                if icommit_order[last_commited_index] == command: in order with the current commit
                    continue
                else:
                    out_of_order = True
                    break
    if out_of_order:
            send((’Done’,’Safety Violated’), to=main)

//Checks for Liveness Violation when round does not progress for majority
function check_progress():
    no_progress = 0
    for validator in {validators U twinValidators}:
        if self.last_executed_round[validator] > config.scenario_max_round:
            no_progress = no_progress + 1
    if no_progress == 2f+1:
        //if 2f+1 ( twins included) had completed, then no possibility of sync up
        send((’Done’,’LIVENESS_VIOLATION’), to=main)
```

```
procedure RECEIVE(msg=('playground', msg, msgtype, round), from_=source):
    handleMsg(source, round, msg, msgtype)
    if self.last_executed_round[source] < round:
        self.msgcount_per_round[source] = 0
    self.last_executed_round[source] = round
    self.msgcount_per_round[source] = self.msgcount_per_round[source] + 1
    if self.msgcount_per_round[source] > 10:
        check_progress();
    check_nocommit_pool(source) //For Potential Liveness Violation
    if round >= config.max_scenario_round:
        check_process_completion() // For Completion of the whole process

procedure RECEIVE(msg=('CommitNotification', round, txnx), from_=source):
    self.last_commited_round[source] = round
    update_nocommit_pool(source) //Update No Commit Pool
    do_safety_check()
    check_process_completion()To check if all validators commited all commands
```

Run.da

```
procedure diemBFT_run():
    .
    // changes for test execution
    validators = new(ValidatorFI, num=(nvalidators+nfaulty)) Twins Included
    for v in [nvalidators+nfaulty]:
            if v in private_keys_validators:
                continue
            private_key, public_key = Cryptography.generate_key()
            private_keys_validators[v] = private_key
            public_keys_validators[v] = public_key
            if v in config['twin']: Same Public/Private Keys for the twin
                twin_id = config['twin'][i]
                t = validators[twin_id]
                private_keys_validators[t] = private_key
                public_keys_validators[t] = public_key
    networkplayground = new(NetworkPlayground)
    setup(networkplayground, scenario, config)
    .
    .
    await(received(('Done',Cause), from_=networkplayground))
    // Cause can be either liveness/safety violation or Completion

procedure main(args):
    self.global_scenarios = readfile(args.scenariogenerated_file_path)
    for scenario in global_scenarios:
        p = new (diemBFT)
        setup(p, scenario) //Initializes All Configs
        start(p)
```

LeaderElection.da

```
function get_leader(round):
    //leaders are precomputed for each scenario and passed via config during setup
    return config.round_leaders[round]
```

Validator.da

```
 function send_message_to_validators(msgtype, msg):
    //all messages within validators needs to be sent to playground with round number
    send(('playground', msg, msgtype, PaceMaker.current_round),
                                    to=config.networkplayground)
def receive((msgtype, msg, source), from_=NP):
    sender = source
    .//Ignore from_ NP as it is always from playground and use source
    .
//Sync Up Logic
def receive(msg=('Proposal', proposal_msg, source), from_=p):
    .
    .
    if diff(proposal_msg.qc, self.blocktree.high_qc) > 1:
        send(('syncup_request', self.blocktree.high_qc), to=p)
    .
    .
def receive(msg=('syncup_request', high_qc, source), from_=p):
    current_block = cached_proposal_block
    response_blocks = []
    while diff(current_block.qc, self.blocktree.high_qc):
        response_blocks.append(current_block)
        current_block = block[current_block.qc.vote_info.id] //parent block
    reverse(response_blocks)
    send(('sync_response', response_blocks),to=p)

def receive(msg=('sync_response', response_blocks, source), from_=p):
    for block in response_blocks:
        self.process_certificate_qc(block)
```

Ledger.da

```
 function commit(block_id):
    .
    //code unchanged
    .
    .
    mempool.check_and_remove(recently_commited_txns)
    //Notify Playground after commiting for Liveliness Check
    send(('CommitNotification', PaceMaker.current_round),
                                to=config.networkplayground)
```