# Phase 3: Design and Pseudo-code for Twins

Team Name: RVP

Raj Patel, 114363611
Prince Kumar Maurya, 114354075
Vishal Singh, 114708875

- Design and pseudo-code for scenario generator
  - Features
    - Operation mode
    - Selection of leaders
    - Enumeration Limits
    - Enumeration Order
    - Partition change triggers
    - Message drops
    - Liveness
  - Design and Pseudocode
    - Network partition generator design
    - Network partition generator pseudo code
    - Scenario generator
      - Design choices for generating scenarios
      - Design
    - Scenario generator pseudo code
- Design and pseudo-code for Scenario Executor
  - Design
  - Network Playground pseudo code
  - Scenario executor pseudo code
- Additional pseudo code
    - Changes in Diem
    - TestRunner
- Contributions

Note for TA: For pseudo code, you may refer to the psuedo_code folder. Additionally, you may use pseudo code extension in vscode to get syntax highlighting.

# Design and pseudo-code for scenario generator

## Features

### Operation mode

The design supports writing test scenarios onto a file in json format as well as supports directly passing the test scenarios to the test executor.

### Selection of leaders

The leader selection can be done deterministically by providing leader assignment for each round or the algorithm does it on a random basis.

### Enumeration Limits

The algorithm limits the enumeration by generating configurations on the fly during scenario generation. For each round assignment, a set of network partitions are chosen and nodes are placed based on the logic discussed in the pseudo-code below.

### Enumeration Order

Enumeration based on partition assignments (deterministic) and randomized is supported.

### Partition change triggers

Partition change triggers are performed by the networking playground based on triggers on the round numbers and the count of messages from a validator.

### Message drops

All forms of message drops are supported by the form of partitions and the partition triggers.

### Liveness

Liveness is ensured by the scenarios generated by the scenario generator. Additionally similar to the paper we introduce three additional rounds in the end with a majority quorum by ensuring supermajority nodes to be part of one of the partitions of the Network. This ensures that the algorithm necessarily performs a block commit.

# Design and Pseudocode

## Network partition generator design

The network partition generator generates all unique ways in which the network can be partitioned across all nodes (including twins).

The algorithm relies on backtracking to generate the partitions.

For example: For a set of 5 nodes the following partitions are generated.
[['_', '_', '_', '_', '_']]
[['_', '_', '_', '_'], ['_']]
[['_', '_', '_'], ['_', '_']]
[['_', '_', '_'], ['_'], ['_']]
[['_', '_'], ['_', '_', '_']]
[['_', '_'], ['_', '_'], ['_']]
[['_', '_'], ['_'], ['_', '_']]
[['_', '_'], ['_'], ['_'], ['_']]
[['_'], ['_', '_', '_', '_']]
[['_'], ['_', '_', '_'], ['_']]
[['_'], ['_', '_'], ['_', '_']]
[['_'], ['_', '_'], ['_'], ['_']]
[['_'], ['_'], ['_', '_', '_']]
[['_'], ['_'], ['_', '_'], ['_']]
[['_'], ['_'], ['_'], ['_', '_']]
[['_'], ['_'], ['_'], ['_'], ['_']]

Note: “_” are just placeholders where a node is placed by the scenario generator when a partition is chosen.

This contains duplicate partitions such as [['_', '_', '_'], ['_', '_']] and [['_', '_'], ['_', '_', '_']]. This is eliminated in the next step which filters the unique partitions.

After performing pruning the following is the set obtained.
[['_', '_', '_', '_', '_']]
[['_', '_', '_', '_'], ['_']]
[['_', '_', '_'], ['_', '_']]
[['_', '_', '_'], ['_'], ['_']]

[['_', '_'], ['_', '_'], ['_']]
[['_', '_'], ['_'], ['_'], ['_']]
[['_'], ['_'], ['_'], ['_'], ['_']]

This set contains all unique scenarios in which the 5 nodes can be divided.

Along with this, the network partition generator also returns a set containing partition with atleast the majority (2 * F + 1) to make the work simpler for the scenario generator. This majority will in turn be used to ensure liveness

These are the majority partition set returned by the network partition.
[['_', '_', '_', '_', '_']]
[['_', '_', '_', '_'], ['_']]
[['_', '_', '_'], ['_', '_']]
[['_', '_', '_'], ['_'], ['_']]

## Network partition generator pseudo code

This can be found under psuedo_code/network_partition_generator

```
Module NetworkPartitionGenerator{
  Procedure generatePartitionRec(
   index, //index of node in the list
   node_list, // the list of nodes
   current_partition, // current partition
   partitions // Final list of all possible partitions
 ) {
   if index == len(node_list):
     partitions <- partitions ∪ current_partition
   else:
     // either include the current value to existing partition
     current_partition[lastElement] <- current_partition[lastElement] ∪ node_list[index]
     generatePartitionRec(index+1, node_list, current_partition, partitions)
     current_partition[lastelement].pop()

     // else create its own new partition
     new_partition=∅
     new_partition <- new_partition ∪ node_list[index]
     current_partition <- current_partition ∪ new_partition
     generatePartitionRec(index+1, node_list, current_partition, partitions)
     current_partition.pop()
 }

 Function prune_duplicate_partition(
   partitions // list of partitions to be pruned
 ) {
   // prune scenarios such as [_, _, _ | _, _] and [_, _ | _, _, _]
   // Sort partition by their partition sizes
   for p in partitions:
     p.sort()

   // converts paritions into [_, _, _ | _, _] and [_, _, _ | _, _]
   // resulting unique partion set
   unique_partitions <- {}

   // iterate over all generate partition
   for p in partitions:
     key <- ∅
     for np in p:
       key <- key ∪ |np|
```

```
    // key formed for example above will be "32"
    if key not in unqiue_partitions
      unique_partitions[key] <- p

  // take only the values
  partitions <- unique_partitions.values()
  return partitions
}

 Function get_partition_scenarios(
  N, // Total Number of nodes considering twins
  F // Number of Faulty nodes
) {
  // generate a node list
  node_list <- [i for i in 0...N]
  // initialize a empty set
  current_partition <- ∅
  // fill up the initial value
  current_partition <- current_partition ∪ [arr[0]]
  // resulting partitions
  partitions <- ∅
  // generate using backtracking
  generatePartitionRec(1, node_list, current_partition, partitions)
   // pruned partitions
  partitions <- prune_duplicate_partition(partitions)
  // get partitions that contain majority
  majority_partitions <- get_majority_partitions(partitions, F)
  // return partitions and majority partitions
  return partitions, majority_partitions
}

Function get_majority_partitions(
    partitions, // list of all pruned partitions
    F // number fo faulty replicas
  ){
    majority_partitions = ∅
    // iterate over all partitions
    for p in partitions:
      // if length of first partition is greater than equal to super majority
      if len(p[0]) >= (2 * F + 1):
        majority_partitions <- majority_partitions ∪ p
```

```
    return majority_partitions
  }
}
```

## Scenario generator

### Design choices for generating scenarios

There are several ways in which the scenario generator can be designed to cater to the requirements stipulated to ensure liveness.

**Choice 1: Scenario generation based on gauging past progress**
In this form of scenario generation, the generator lays out scenarios in which it gauges the next network partition based on the progress of the past network partition and the state of Diem processes inferred from messages. We call this type of choice to be **memory full**. In which some reasoning is required based on the previous state to determine the next state.

Consider the following choice of network partition for round 1 (R1) with the leader as node 1.
R1 :
<u>1</u>, 2 | 4, <u>1’</u> | 3
To maintain progress some consensus has to be there such that nodes can transition to the next round. Let us choose one such network partition that provides a consensus state. This network partition will be chosen by some trigger.

R1 :
<u>1,</u> 2 | 4, <u>1’</u> | 3
<u>1,</u> 2, 3 | 4 | <u>1’</u> -> consensus 1

Now nodes 1, 2, 3 will progress to the next round. For 1’ and 3 to progress as well, we introduce one more network scenario such that progress gets maintained.

R1:
1, 2 | 4, 1’ | 3
1, 2, 3 | 4 | 1’
1’, 4, 2 | 1, 3 -> progress for remaining nodes consensus 2

R2:
1’, 4, 2 | 1, 3 -> node 2 should be consistent

In cases where commits have to be performed, leader assignments have to be tracked as well.

Advantages :

1. This type of scenario generation keeps the scenario executor's implementation lean. The scenario executor just has to choose a network partition based on some state trigger and the round number of the current validators.
2. With past progress in track, specific scenarios can be generated to ensure block commits are deterministic.

Disadvantages:

1. Scenario generation becomes complicated. Gauging the next state based on the previous state is difficult. (A **DFA (Deterministic finite automata)** can be produced to keep track of state transition which could lower down the complexity).

**Choice 2: Memoryless scenario generation**

In this form of scenario generation network partitions can be laid out randomly while ensuring progress. In Diem's context a quorum is required to maintain progress hence in each round there has to be a consensus state which could drive a QC or a TC. Consider the example in choice 1. For ensuring progress for some nodes only the first consensus state is required. The scenario executor can then define partition change in such a way that progress is maintained for remaining nodes that were not part of consensus (4, 1'). This also allows for random leader assignments as even if the leader is in a minority partition progress is ensured by forming a TC.

Advantages:

1. Implementation of scenario generation is simple.

Disadvantages:

1. Block commits are difficult to be determined based on scenarios.
2. The scenario executor has to cater to ensuring the progress of non-minority nodes in some scenarios.

## Design

Our design extends choice 2 discussed above. For block commits to happen, in the end, we provide 3 rounds having a scenario with the supermajority that would ensure a block commit. This would ensure the liveness property holds of the algorithm. For any safety violation to happen it would have happened prior to this point. Likewise, if the algorithm misses committing after this generated scenario it would flag a liveness violation.

The ScenarioGenerator, generates scenario in a recursive pattern. In a given round the first M - 1 network partitions are meant for finding safety / liveness violations while the last round is meant to perform some progress.

Example :

Round 1 :
1…. M -1 random partitions meant for finding safety / liveness violations
Mth partiton with majority consensus to maintain progress by forming QC or TC
...
...
...

Round N :

1…. M -1  random partitions meant for finding safety / liveness violations
Mth partition with majority consensus to maintain progress by forming QC or TC

Round N + 1 :
Majority consensus to ensure liveness

Round N + 2:
Majority consensus to ensure liveness

Round N + 3 :
Majority consensus to ensure liveness

The last three additional rounds give chance for any BFT algorithm (Diem in our case) to commit a block.

**Assign leaders**
Assign leaders provide leader assignments for each round except for the last three rounds where we provide scenarios for block commit to happen. Assignments can be provided for leaders to be chosen deterministically. It also supports additional types such as sequential where leader assignments happen in a round-robin fashion similar to the way in PBFT.

**Round assignment**
The round assignment first chooses the number of partitions to be present at each round. It chooses a random number with the help of M (max partition per round). Hence, the partition per round is between [1, M] (inclusive).
Next, it uses NetworkPartitionGenerator to generate all unique ways for partitioning N nodes. Now, for each round and each partition in that round until the penultimate partition (for that specific round), it chooses a network partition and uses a populate partition to make placement for the leader as well as other nodes in that partition.

For the final assignment, it chooses the one which forms a majority from the majority partition set and performs node placement using a populate consensus partition. This ensures that the last partition will be used for subsequent messages such that it forms a QC or TC to advance to the next round and maintain the liveness of the system.

In the end, it generates three-round scenarios which can form a quorum and can perform a block commit.

## Scenario generator pseudo code

This can be found under psuedo_code/scenario_generator

```
Module ScenarioGenerator{
 N // total number of rounds
 total_nodes // total nodes excluding twins
 M // max partitions per round
 F // number of faulty nodes or twins
 probability_of_overlap // probability of overlapping partition
 probability_partition_has_overlap // probability that a network partition contains a overlap
 twin_nodes // set of nodes that are twins

 Function round_assignment(
   leader_assignments,
   twin_nodes,
   parition_assignments
 ) {
   // generate the number of partition per round
   partition_choices <- [i for i in 1...M + 1]
   partition_per_round <- ∅

   for i in 1...(N + 1)
     partition_per_round[i] <- random.choice(partition_choices)

   // generate all network partition scenarios from sum of total nodes and its twins
                                    partition_scenarios,            major_partitions            <-
NetworkPartitionGenerator.get_partition_scenarios(total_nodes + F)
   final_assignments <- ∅

   for i in 1...(N + 1):
     // if assignment is deterministic (user defined)
     if i in parition_assignments:
       final_assignments[i] <- parition_assignments[i]
       continue

     // partitions can be populated randomly expect for last partition
     for j in 0..(partition_per_round[i] - 1)
       // generate and populate partition
       final_assignments[i] <- final_assignments[i] ∪ (populate_partition(
         random.choice(partition_scenarios),
         ))

     // ensuring that the last partition trigger will have super majority
```

```
    final_assignments[i] <- final_assignments[i] ∪ (populate_conensus_partition(
      random.choice(major_partitions),
      leader_assignments[i],
      twin_nodes
    ))

  // adding three extra rounds with super majority to ensure liveness
  majority = random.choice(major_partitions)
  // non faulty are always placed first hence the leader choosen down below
  // at random will always be selected
  majority = populate_conensus_partition(
    majority,
    random.choice(non_faulty),
    twin_nodes
  )
  // set of non_faulty nodes
  non_faulty = set([i for i in 0...(total_nodes + F)]) - twin_nodes
  // add three additional nodes
  for i in 0..3:
    leader <- random.choice(non_faulty)
    final_assignments[N + i]<- final_assignments[N + i] ∪ majority
    // add the leader for the additional round
    leader_assignments[N + i] <- leader

  return final_assignments
}

Function populate_partition(network_partition) {
  // all possible nodes including twins
  nodes <- [i for i in 0...(total_nodes + F)]
  // shuffle the node set
  node_shuffled <- random.sample(nodes, total_nodes + F)

  // count of partitions in specific network partition
  no_of_partitions <- |network_partition|
  i <- 0
  k <- 0

  while i < no_of_partitions:
    //size of that partition
    partition_size <- |network_partition[i]|
    // populate each partition subset
```

```
    j <- 0
    while j < partition_size:
      network_partition[i][j] <- node_shuffled[k]
      j <- j + 1

    k <- k + 1

  // add overlaps based on probability
  // for a partition choose if overlap should be added between its partitions
  if random.uniform(0, 1) < probability_of_overlap:
    // for each partion in network partition
    for i in 0...|network_partition|:
      // check probability of partition to have overlap
      if random.uniform(0, 1) < probability_partition_has_overlap:
        // choose a node which is not part of the current partition
        non_partition_nodes <- set(node_shuffled) - set(network_partition[i])
        // add new overlapping node to existing partition
        network_partition[i] <- (network_partition[i] ∪ list(random.choice(non_partition_nodes)))

  return network_partition
}
 Function populate_conensus_partition(
  network_partition,
  leader //
  twin_nodes //number of twin nodes
){
  // generate nodes including twins
  nodes <- [i for i in 0..(total_nodes + F)]
  // shuffle the generated set
  nodes_shuffled <- random.sample(nodes, total_nodes + F)

  // if all nodes have to be in the partition then return
  if len(network_partition) == 1
    return nodes_shuffled

  // fill the majority partition first
  majority_partition <- network_partition[0]
  // place the leader in majority partition
  majority_partition[0] <- leader
  // remove leader from nodes shuffled
  nodes_shuffled.remove(leader)
  // remove leader from twin nodes if twin is leader
```

```
  twin_nodes.remove(leader)
  // remove twin nodes from node shuffled
  nodes_shuffled <- nodes_shuffled - twin_nodes

  i <- 1
  // first place the non twin nodes
  // if nodes are exhausted then place the twin nodes
  while i < len(majority_partition) && nodes_shuffled ≠ ∅
    select_node <- nodes_shuffled.pop
    majority_partition[i] <- select_node

    if nodes_shuffled = ∅
      // if shuffle nodes become empty now place the twin nodes
      nodes_shuffled <- nodes_shuffled + twin_nodes
      // make twin nodes as null so that it is not added twice
      twin_nodes = ∅

    i <- i + 1

  // if twin nodes have not been previously added, add twin nodes to the node_shuffled set
  nodes_shuffled <- nodes_shuffled + twin_nodes
  // fill up remaining partitions
  for i in 1....|network_partition|
    for j in 0...|network_partition[i]|
      network_partition[i][j] <- nodes_shuffled.pop

  return network_partition
}

Function assign_leaders(
  type = "random"
  assignments = ∅
) {
  if !(|assignments| <= N and max{assignments} <= N)
    return ⊥

  // non deterministic assigments to be assignmed
  pending_assignments = ∅
  // final pool of assignments
  final_assignments = ∅

  // finding pending assignments
```

```
  for i in 1...(N + 1):
    if i ∉ assignments:
      pending_assignments <- pending_assignments ∪ i

  // checking for case of sequential
  if type == "sequential":
    // go from round 1  to N
    for i in 1....(N + 1):
      if i ∉ in assignments:
        final_assignments[i] <- i % total_nodes
      else:
        final_assignments[i] <- assignments[i]
  // checking for case of random
  else if type == "random":
    // go from round 1 to N
    for i in 1....(N + 1)
      if i ∉ assignments:
        // choose a random node between 0 and total_nodes
        final_assignments[i] <- random.randint(0, total_nodes - 1)
      else:
        final_assignments[i] <- assignments[i]
  else:
    return ⊥

  return final_assignments
 }
}
```

# Design and pseudo-code for Scenario Executor

## Design

The scenario executor based on the mode of operation of the scenario generator either reads the json file of the configuration or else directly receives the configuration. The scenario executor performs the task of spawning the validators, initialization of the network playground, running the scenarios, and performing verification and validation for safety and liveness.

**Spawning the validators**

Spawning of validators is done similar to the way it is done in phase 2 of the implementation. Once, the spawning of 3 * F + 1 validator is done. It further spawns additional F nodes with configuration as i % (3 * F +

1). Configuration means the cryptographic key which is used uniquely identifies the validator. While spawning it also provides the leader assignments to each node so that the `getLeader` method overridden in the validator can make use of the leader assignments provided per round.

**Network playground**
The network playground is the glue that ties in the entire execution logic and ensures progress for nodes that belong to a minority in a particular round. In order to do this, the selection of network partitions is based on the highest seen round by the validators. Meaning if a message sent by a higher round node is received at a node in the lower round then in-order for progress the network partition for the lowered ordered node is chosen based on the round number of the higher round node. This is illustrated in the example below :
R1 : 1, 2, 3 | 4, 1'
R2 : 2, 1', 4 | 3, 1

In round 2 when 2 sends a message to 1' and 4. Although, 1' and 4 belong to a prior round (Round 1 in our case). The network playground will choose round 2 as the network partition as the proposal message is sent for R2 and not R1. Similar logic would tie in whenever other types of message exchanges are being done. Hence, whenever differing round scenarios are encountered the network playground would ensure that progress happens.

Another setting that network playground has to handle is partition change in a given round itself. We perform this by the count of messages for a particular round. Formally, we maintain a state that keeps track of messages passed by each validator at each round. If the message passed exceeds the available partitions at a round we pick the last partition which will be the majority partition that will ensure progress from that round. For example :

R1 : 1, 2 | 3, 4 | 1'
1, 2, 1' | 3, 4
1, 2, 3 | 4, 5

In the example above when the first message is passed for that round view (the round view is the highest seen round for that message) then it will choose the first partition, next for the second message it will choose the next partition and for subsequent messages, the last partition is chosen.
This will make sure that there are some sets of nodes that will make progress for sure. And the subsequent nodes can eventually make progress.

**Running the scenarios**
For running the scenarios the network playground will drive the communication and transition. Thus while running, the round number passed by the messages can be used to interpret the termination state. More formally, when the round number of any message exceeds R + 3 (total rounds defined + three extra rounds) then we trigger termination of nodes.

**Verification**

Checking for safety violations
For checking safety violations one can check the ledger state to see for any differing commits. Differing commits are indicative of a bug in the algorithm which allowed such commits to happen.
Other safety violations can lead to no blocks to commit which would be indicative of a liveness violation.

Checking for liveness violations
For checking liveness violations one can check the ledger state to see if consistent block commits are present at atleast 2 * f + 1 nodes (excluding the other twin node when checking for consistency). Then the ledger state is consistent. If in case the scenarios are laid out such that a block commit should happen, then the same has to be reflected in the ledger. No block commit in such a scenario is indicative of liveness violation.

Liveness also implies progress hence the logs must contain incremental changes in round numbers along with the ledger states.

There are two approaches for verification and validation.

1. Online
2. Offline

Online verification: In online verification, the scenario executor performs verification and validation on the fly. In this, it checks for any safety and liveness violation based on the ledger and logs generated.

Advantages :

1. Any safety or liveness violation can be detected immediately.
2. It usually reduces the execution time by detecting bugs early.

Disadvantages:

1. Could be computationally expensive as it makes periodical checks.

Offline verification: In offline verification, the scenario executor performs liveness and safety checks at the last when the execution is either done or the execution exceeds a certain threshold. The threshold is a guesstimate of how long it takes for a set number of rounds to execute. (As there are no network delays this can be estimated quite accurately by taking the upper bound for each timeout to occur multiplied by the total number of rounds).

Advantages:

1. Would be relatively less computationally expensive.
2. Simple to implement.

Disadvantages:

1. Execution time is relatively higher as it waits for the upper bound time to reach in case of a liveness bug.

In our design, we employ offline verification as it is much more simple to implement.

## Network Playground pseudo code

This can be found under pseudo code/network_playground

```
// acts as central hub for message interception
Module NetworkPlaygroud{
 Partition_Config, // List of Partition configurations for each round
 number_of_nodes, // count of total number of nodes excluding twins (3 * F + 1)
 total_nodes, // count of nodes excluding twin (4 * F + 1)
 twin_nodes  // set of which nodes are twin
 broadcast <- -1
 max_round <- 0
 // the node_states kepps track of the count of each message for each sender
 // used for determining the network partition for a given round
 // is incremented whenever a message from a sender for a round is received
 node_states <- {} //dictionary with node as key and state as value
  Func Main() : EventLoop{
     loop : Wait for next event M; Main.start_event_processing(M)
     Procedure start_event_processing(M) {
         onReceive(M)
     }
 }

 // event handler
 Procedure onReceive(M)
 {
   // get the sender id
   senderID <- M.senderID
   // get the receiver id
   receiverID <- M.receiverID
   num_nodes <- number_of_nodes
   ScenarioExecutor.OnReceive(M)
   // if the receiver is to broadcast message
   if(receiverID == broadcast)
     for nodeID in num_nodes
       // send message to both the receiver and its twin which will be 3 * f + 1 + receiverID as
per our logic
       twinNodeID <- nodeID + number_of_nodes
       // same partition send message to receiver
       if same_partition(senderID, nodeID)
         send(M, M.payload, nodeID)
       // same partition then send message to receiver's twin
       if same_partition(senderID, twinNodeID)
```

```
        send(M, M.payload, twinNodeID)
  else
    twinReceiverID <- M.receiverID + number_of_nodes
    if same_partition(senderID, receiverID)
      send(M, M.payload, receiverID)
    if same_partition(senderID, twinReceiverID)
      send(M, M.payload, twinReceiverID)

  // increment the node state for the sender and the round
  node_states[senderID][round] <- node_states[senderID][round] + 1
 }
 Function same_partition(
  M, //Message for which partition should be checked
  sender //sender of message
  receiver //receiver of message
  ) {
    if receiver > total_nodes
        return False

    round <- M.round
    // choose the network-partition set for the given round
    current_partitons <- Partition_Config[round]
    if node_states[sender][round] != ∅:
        // choose the partition based on min of the count of messages for that round
        // or the size of partition set
        partition_id <- min{ node_states[sender][round], |current_partitons| - 1 }
    else:
       // round encountered for the first time, choose the very first partition
        node_states[senderID][round] = 0
        partition_id <- 0

    // check if the sender and receiver belong to the same parititon
    for partition in current_partitons[partition_id]:
        if (sender ∈ partition) && (receiver ∈ partition):
            return True

    return False
  }
}
```

## Scenario executor pseudo code

This can be found under psuedo_code/scenario_executor

```
Module ScenarioExecutor{
num_nodes, //total number of nodes excluding twin
twin_nodes, //nodes for which to create twins
round_partitions, //Configuration of Partition for each round
round_leaders //Leader for each round
num_of_rounds //number of rounds for which the network playground should run
id // file id of Configuration
mode // mode of operation
timeout // some timeout for the scenario executor
reached_timeout <- False

// if some id is given load the configuration
if id != None:
 round_partitions, round_leaders = load_from_json(id)

current_round <- 0
 //Initialise Network Partition

 Function start()
   // if some id is given load the configuration
   // this indicates offline mode where reading config is required
   if id != None:
     round_partitions, round_leaders = load_from_json(id)
    NetworkPlaygroud <- NetworkPlayground(round_partitions, number_of_nodes)
   F <- |twin_nodes| / 2
   Keys <- generateKeys(num_nodes)
   Nodes <- ∅

   //Run the Algorithm for Number of round
   np <- NetworkPlayground()
   //We need to pass Partition Con
   for i in 1...(num_nodes+F):
   //This assigns same cryptographic keys to twins
     Nodes <- spawn(i, Keys[i % num_nodes], round_leaders, np)

   // start a timer
   start_timer(timeout, onTimeout)
   //Wait till we receive a callbacks from Network Playground with round exceeding the limit
   await (current_round<=num_of_rounds + 3 and !reachedTimeout);
```

```
  for i in 1...(num_nodes+F):
    // terminate all nodes as all rounds has been done
    terminate(i)

OnReceive(M){
  current_round=M.round;
}

onTimeout() {
  reachedTimeout <- True
}

Function verify(){
  SafetyCheck() &&
  LivenessCheck()
}

Function SafetyCheck(){
  //check if there are differing commits
  for node in num_nodes:
    ledger = ∅

    if node in twin_nodes:
      // condition required so that twin ledger is also included
      twin = node + total_nodes
      // returns ledger as a sequential list
      twin_ledger = getLedger(twin)
      ledger = twin_ledger

    // choose any one of complete ledger which is ledger or twin_ledger
    ledger = getLedger(node) or ledger

    if ledger == ∅:
      continue

    count += 1
    for other_node in num_nodes:
      if node != other_node:
        other_ledger = getLedger(node)
        if other_ledger == ∅:
          continue
```

```
        // perform sequential check for each block commited
        // order as well as block contenets are verified
        if ledger != other_ledger:
          // ledger are not consistent
          return False

  if count < (2 * |twin_nodes| / 2) + 1:
    // less than 2 * f + 1 nodes commited
    // this will also do a liveness check here itself although
    // we have defined another explicit function for this check
    return False

  return True
}


Function LivenessCheck(){
  //check if atleast be one commited block for every node
  count == 0
  for node in num_nodes:
  // returns ledger as a sequential list of block commits
    ledger = node.getLedger()

    if node in twin_nodes:
      // condition required so that twin ledger is also included
      twin = node + total_nodes
      twin_ledger = getLedger(twin)
      // choose any one of complete ledger which is ledger or twin_ledger
      ledger = twin_ledger or ledger

    //if ledger has content increment
    if |ledger| != 0:
      count += 1

  // check if count is greater than or equal to 2 * f + 1
  return count >= (2 * |twin_nodes| / 2) + 1
}
}
```

# Additional pseudo code

## Changes in Diem

Can be found under psuedo_code/diem

```
Module Diem {
id
keys
leader_assignments
NetworkPlaygroudNode // reference of network playground for relaying messages
broadcast <- -1

Module LeaderElection{
 leader_assignments // dictionary of leader assignments

  Function get_leader(round) {
   return leader_assignments[round]
  }
}

Module Main {
 // struct for message
 Message {
  payload,
  round,
  sender,
  receiver
 }

 // this procedure has to be used when sending any messages from diem
 Procedure sendMessage(payload, round, receiver) {
  // payload contains the message payload
  // round contains the max of round of original message or current round
  // receiver is the receiver (could be a node or broadcast type)
  M <- Message(payload, max(round, PaceMaker.current_round), id, receiver)
  send(NetworkPlaygroudNode, M)
 }
}

Module Mempool{
  current_transaction = 0
 //The get_transaction method is overidden to get propose dummy blocks
```

```
 Procedure get_transactions() {
   // payload contains the message payload
   // round contains the max of round of original message or current round
   // receiver is the receiver (could be a node or broadcast type)
   current_transaction <- current_transaction + 1
   return str(id) + "-" + str(current_transaction)
  }
 }
}
```

## TestRunner

Can be found under psuedo_code/test_runner

```
Module TestRunner {
 mode // operation mode offline / online
 configurations // defines the config set

 Function RunTests() {
   for config in configurations:
     generator = ScenarioGenerator(
       config.N, // total number of rounds
       config.total_nodes // total nodes excluding twins
       config.M // max partitions per round
       config.F // number of faulty nodes or twins
       config.probability_of_overlap // probability of overlapping partition
       config.probability_partition_has_overlap // probability that a network partition contains a
overlap
       config.twin_nodes // set of nodes that are twins
     )
     leaders = generator.assign_leaders(config.type, config.assignments)

     round_assignments = generator.round_assignment(
       leaders,
       config.twin_nodes,
       config.parition_assignments
     )

     id = None
     if mode == "offline":
       id = save_to_json(leaders, round_assignments)

     // although we are calling it immediately this can be called at a later point
```

```
     // emphasis being a file is saved by save_to_json with name as id
     // this file can be picked up at anytime by scenario executor
     executor = ScenarioExecutor(
       config.total_nodes,
       config.twin_nodesleaders,
       round_assignments,
       leaders,
       config.N,
       id,
       mode
     )
     executor.start()
     executor.verify()
 }
}
```

# Contributions

Pseudo codes :
Raj Patel, 114363611 Scenario generator
Prince Kumar Maurya, 114354075 -  Network Playground
Vishal Singh, 114708875 - Scenario Executor

The design, design decisions and documentation were done mutually by all team members.