**Implementation of Twins and Testing of diemBFT**

**Test Generator:**

**Config file containing the test parameters for the test generator contains the following:**

| "N\_validators, *// Number of validators*  "N\_partitions", *// Number of non-empty partition*s  "Allowed\_leader\_type", *// Type of leader, “FAULTY”, “NON-FAULTY”, “ALL”.*  "N\_rounds"*, // Number of rounds.*  "is\_deterministic", *// Whether to generate test cases sequentially or randomized.*  "Is\_with\_replacement", *// Whether to enumerate with replacement when permuting scenarios over n\_rounds.*  *// Only used when is\_deterministic is False*  "N\_testcases", *// Number of test cases to be generated.*  "Test\_file\_batch\_size", *// Number of test-case in each test case file.*  "N\_twins", *// Number of twins.*  "Intra\_partition\_drop\_types", *// Subset of {Proposal, Vote, Timeout}, list of message types that can be dropped intra-partition*  *// We allow a maximum of 2 message types to be dropped. If more than 2 are given, we take the first 2 in the list*  "seed": 12345 |
| --- |
|  |

**Helper functions for each step of the test generator are as follows:**

* **Step 1:** get\_all\_possible\_partitions()
* **Step 2:** generate\_leader\_partitions()
  + Step 2.1 : enumerate\_leader\_partitions\_with\_drops()
* **Step 3:** enumerate\_leader\_partition\_pairs\_over\_rounds()

**STEP - 1:**

Generate all possible partition scenarios based on the .

| *// Returns all possible ways the list of validator IDs can be divided into n\_partitions.*  *//"generate valid partition" indicates that the generated partitions will eventually assure the presence of quorum.*  Procedure get\_all\_possible\_partitions(validator\_ids, n\_partitions, generate\_valid\_partition, num\_faulty):  all\_possible\_partitions <- []  t = int(count\_part(len(validator\_ids), n\_partitions))  for i in range(t):  x <- gen\_part(validator\_ids, n\_partitions, i)  if generate\_valid\_partition:  *// isValidPartition() will check if in the generated partition quorum can exist.*  if isValidPartition(x, num\_faulty):  all\_possible\_partitions.append(x)  else:  all\_possible\_partitions.append(x)  return all\_possible\_partitions |
| --- |

**STEP - 2:**

Generate all the possible leader-partition pairs.

Args:

* all the partition scenarios,
* validator ids: id of all the validators including twins,
* twin\_ids: id of all twin validators
* leader\_type: variable to control the type of leader in each round.

| Procedure generate\_leader\_partitions(partition\_scenarios, validator\_ids, twin\_ids, leader\_type):  leader\_partitions\_pairs <- []  *//contains a list of all validator ids except twins i*ds.  all\_validator\_ids <- set()   [all\_validator\_ids.update(partition) for partition in partition\_scenarios[0]]  *//faulty\_validators\_id\_list contains faulty validator id.*  faulty\_validators\_id\_list <- [validatorId.split("\_")[0] for validatorId in all\_validator\_ids if len(validatorId.split("\_")) > 1]  *// Assigning leader in each subset of partition depending on the type of leader.*  for partition in partition\_scenarios:  for validator\_id in validator\_ids:  if leader\_type == "ALL":  leader\_partitions\_pairs.append((validator\_id, partition))  elif leader\_type == "FAULTY" and validator\_id in twin\_ids:  leader\_partitions\_pairs.append((validator\_id, partition))  elif leader\_type == "NON-FAULTY" and validator\_id not in faulty\_validators\_id\_list:  leader\_partitions\_pairs.append((validator\_id, partition))   return leader\_partitions\_pairs |
| --- |

**STEP - 2.1:**

Combine leader-partition pairs with every kind of intra-partition message drop type i.e. Vote, Proposal, or Timeout.

| Procedure enumerate\_leader\_partitions\_with\_drops(leader\_partitions, drop\_types):  drop\_types <- drop\_types[:2]  result <- []  drop\_scenarios <- [[]]   *// no drop scenario.*  if len(drop\_types) > 0:  drop\_scenarios.append(drop\_types)   *// generate all the possible drop combinations*  if len(drop\_types) > 1:  drop\_scenarios += [[drop\_type] for drop\_type in drop\_types]   for leader, partition in leader\_partitions:  for drop\_scenario in drop\_scenarios:  result.append((leader, partition, drop\_scenario))  return result |
| --- |

**STEP - 3:**

Combine rounds with leader-partition pairs with or without replacement and write the required number of test cases in a file in a batch of size “batch\_size”.

| Procedure enumerate\_leader\_partition\_pairs\_over\_rounds(leader\_partition\_pairs, n\_rounds, n\_testcases, is\_deterministic,is\_with\_replacement, validator\_twin\_ids, n\_validators, batch\_size):   round\_leader\_partition\_pairs <- []  total\_leader\_partition <- len(leader\_partition\_pairs)  index\_list <- list(range(total\_leader\_partition))    *// Combine rounds with leader partition pair in a deterministic way without replacement.*   if not is\_with\_replacement and is\_deterministic:  all\_round\_combinations <- []   for each\_combination in list(itertools.combinations(index\_list, n\_rounds)):  all\_round\_combinations += list(itertools.permutations(each\_combination))  if len(all\_round\_combinations) > n\_testcases:  break   else:  if is\_deterministic:  permutations <- [[] for \_ in range(total\_leader\_partition \*\* n\_rounds)]  all\_round\_combinations <- permutations\_with\_replacement(total\_leader\_partition, n\_rounds,  permutations)  else:  all\_round\_combinations <- enumerate\_randomized(leader\_partition\_pairs, n\_rounds, n\_testcases)   count\_testcases <- 0  flag <- False   for permutation in all\_round\_combinations:  round\_leader\_partition\_pairs.append(  accumulate(permutation, leader\_partition\_pairs, n\_rounds, validator\_twin\_ids, n\_validators))  count\_testcases += 1   if count\_testcases == n\_testcases:  flag <- True   if not flag and len(round\_leader\_partition\_pairs) == batch\_size:  dump\_file(round\_leader\_partition\_pairs, count\_testcases)  round\_leader\_partition\_pairs <- []   elif flag and round\_leader\_partition\_pairs:  dump\_file(round\_leader\_partition\_pairs, count\_testcases)  break   *// Enumerate n\_test\_cases ways of randomly arranging leader\_partition pairs over n\_rounds* Procedure enumerate\_randomized(leader\_partition\_pairs, n\_rounds, n\_test\_cases):  return [[random.randrange(len(leader\_partition\_pairs)) for \_ in range(n\_rounds)] for \_ in range(n\_test\_cases)]   *// Return all ways of arranging n leader\_partition pairs over k round*s Procedure permutations\_with\_replacement(n, k, permutations):  m <- 0  if k < 1:  return permutations   for i in range(27):  permutations[i].append(m % n)  if (i % n \*\* (k - 1)) == n \*\* (k - 1) - 1:  m <- m + 1   return permutations\_with\_replacement(n, k - 1, permutations)  *// Convert given testcase into a JSON object* Procedure accumulate(index\_list, leader\_partition\_pairs, n\_rounds, validator\_twin\_ids, n\_validators):  round\_leader\_partitions <- [leader\_partition\_pairs[idx] for idx in index\_list]  curr\_test\_case <- JsonObject(n\_validators, n\_rounds, validator\_twin\_ids, round\_leader\_partitions)   return curr\_test\_case.toJSON()  *// Writes the list of JSON elements to file* Procedure dump\_file(elements, file\_count):  file\_name: str <- "../testcases/testcases\_batch\_" + str(file\_count) + ".jsonl"   with open(file\_name, "wb") as outfile:  for element in elements:  if element is not None:  outfile.write(element.encode() + b"\n") |
| --- |

**Execution point of test generator.**

| if 'seed' in generator\_config: random.seed(generator\_config['seed'])   *// Generate all the validator ids.*  validator\_ids <- [validator\_id for validator\_id in range(generator\_config['n\_validators'])]  *// Generate all twin validator ids based on the number of validators.*  twin\_ids <- [validator\_id for validator\_id in range(generator\_config['n\_twins'])]   validator\_and\_twin\_ids <- [str(validator\_id) for validator\_id in validator\_ids] + [str(validator\_id) + "\_twin" for validator\_id in twin\_ids]    *// Step 1: generate all possible partition scenarios.*  partition\_scenarios <- generate\_partitions.get\_all\_possible\_partitions(  validator\_and\_twin\_ids, generator\_config['n\_partitions'])    *// Step 2: generate all possible leader-partition pairs.*  leader\_partition\_pairs <- generate\_leader\_partitions(  partition\_scenarios, validator\_ids, twin\_ids, generator\_config['allowed\_leader\_type'])    *// Step 2.1: Adding message drops*  leader\_partitions\_with\_drops <- enumerate\_leader\_partitions\_with\_drops(  leader\_partition\_pairs, generator\_config['intra\_partition\_drop\_types'])   *// Step 3: generate test-cases by combining rounds with leader partition pairs in either deterministic or randomized ways.*  enumerate\_leader\_partition\_pairs\_over\_rounds(leader\_partition\_pairs=leader\_partitions\_with\_drops, n\_rounds=generator\_config['n\_rounds'], n\_testcases=generator\_config['n\_testcases'], is\_deterministic=generator\_config['is\_deterministic'], is\_with\_replacement=generator\_config['is\_with\_replacement'], validator\_twin\_ids=twin\_ids, n\_validators=len(validator\_ids), batch\_size=generator\_config['test\_file\_batch\_size']) |
| --- |

**Test Executor :**

The Test Executor is the module with which the user primarily interacts in order to run test cases. The user passes a list of TestConfig objects as a JSON file. Each TestConfig object describes a test case that needs to be executed by the DiemTwins testing framework.

#### **Design assumptions**

Here, we briefly list out the design decisions/assumptions taken in addition to the ones presented in the Twins paper and the phase 3 document.

* We let each validator run for n\_rounds + 3 rounds. Here, the first n\_rounds will be simulated as per the TestCase generated by our Test Generator module. We will then allow the system to execute 3 more rounds without any network partitions. This will allow all non-faulty replicas to commit any pending blocks, and bring their ledgers up to date.

| class TestExecutor(process):  Procedure setup(test\_case, test\_id, test\_file\_id):  self.n\_rounds <- test\_case.n\_rounds  self.n\_validators <- test\_case.n\_validators  self.n\_faulty <- (self.n\_validators - 1)//3  self.delta <- test\_case.delta  self.config\_id <- str(test\_file\_id) + "\_" + str(test\_id)   Procedure run():  start\_time <- time.time()  private\_keys\_validators <- []  public\_keys\_validators <- []  for validator\_id in range(self.n\_validators):  private\_key, public\_key <- Cryptography.generate\_key()  private\_keys\_validators.append(private\_key)  public\_keys\_validators.append(public\_key)   for validator\_id in range(self.n\_validators):  if validator\_id not in test\_case.twin\_ids:  Procedure ault\_leader <- validator\_id  break   validator\_config <- {  "config\_id": config\_id,  "nrounds": self.n\_rounds,  "nvalidators": self.n\_validators,  "nfaulty": self.n\_faulty,  "delta": self.delta,  "leaders": [leader\_partition.leader for leader\_partition in test\_case.leader\_partitions],  "Procedure ault\_leader": Procedure ault\_leader  }   validator\_map <- dict()   for validator\_id in range(self.n\_validators):  *// Create a validator and add to validator\_map. If its not a twin, its twin\_id is simply the validator\_id*  validator <- new(ValidatorTwins, num=1)  twin\_id <- str(validator\_id)  validator\_map[twin\_id] <- validator   if validator\_id in test\_case.twin\_ids:  *// Create the twin validator process and also add it to the validator map*  validator <- new(ValidatorTwins, num=1)  twin\_id <- str(validator\_id) + "\_twin"  validator\_map[twin\_id] <- validator   all\_validators <- set()  for twin\_id, validator in validator\_map.items():  validator\_id <- int(twin\_id[0])  setup(  validator,  (validator\_config, validator\_id, private\_keys\_validators[validator\_id],  public\_keys\_validators, twin\_id, test\_case, validator\_map))  all\_validators <- all\_validators.union(validator)   start(all\_validators)  await(each(v in all\_validators, has=received(('Done',), from\_=v)))  twin\_ids <- [twin\_id for twin\_id in validator\_map]  *// Orchestrator is to maintain all the executor processes to run multiple test cases.*  class Orchestrator(process):  Procedure setup():  Pass  Procedure run():  start\_time <- time.time()   *// Get list of all test files*  test\_files <- sorted(glob.glob(self.test\_directory + "\*.jsonl"), key=lambda file\_name: (len(file\_name), file\_name))   *// Read and execute each test file one at a time*  for test\_file\_no, test\_file\_name in enumerate(test\_files):  with open(test\_file\_name, 'r') as test\_file:  test\_cases <- [TestCase(  n\_rounds=json\_obj.n\_rounds,  n\_validators=json\_obj.n\_validators,  leader\_partitions=[LeaderPartition(round\_leader\_partition) for round\_leader\_partition in  json\_obj.round\_leader\_partitions],  twin\_ids=json\_obj.twin\_ids,  delta=0.1  )  for json\_obj in [  json.loads(json\_string, object\_hook=JsonObject.object\_decoder)  for json\_string in test\_file.readlines()]]   Executors <- set()   for test\_id, test\_case in enumerate(test\_cases):  executor <- new(TestExecutor)  setup(executor, (test\_case, test\_id, test\_file\_no))  start(executor)  executors.add(executor)   await(each(executor in executors, has=received(('Done',), from\_=executor)))  time.sleep(1)  Procedure main():  orchestrator <- new(Orchestrator)  setup(orchestrator, ("../testcases/",))  start(orchestrator) |
| --- |

**Safety Module:** To ensure safety property, check if the orders of the transactions committed in the ledger file of all the validators are the same or not. If it’s matching, safety property is ensured else not.

| Procedure check\_safety(validator\_ids):  for validator\_id\_a in validator\_ids:  for validator\_id\_b in validator\_ids:  for i in range(min(len(ledger\_a\_lines), len(ledger\_b\_lines))):  if ledger\_a\_lines[i] != ledger\_b\_lines[i]:  return False  return True |
| --- |

**Liveness Module:** To ensure the liveness property a “no-op” transaction is sent at the end of all the transactions. If the “no-op” transaction is committed in the ledger file of all the validators, then the Liveness property is ensured.

| Procedure check\_liveness(validator\_ids):  for validator\_id in validator\_ids:  if last\_ledger\_line\_validator != 'no-op':  return False  return True |
| --- |

**Network playground:**

| class ValidatorTwins(process, Validator):  Procedure send(payload, to):  to\_validator\_ids <- to  message\_type, message <- payload   to\_validator\_twin\_i*ds // contains the list of twin validator ids if the destination validator id is having twin.*   if not should\_partition(message\_type, self.pacemaker.current\_round):  *// Send the message to all recipients without applying filtering logic*  for to\_validator in [validator\_map[to\_twin\_id] for to\_twin\_id in to\_validator\_twin\_ids]:  super().send(payload, to=to\_validator)  else:  *// Only send to the twin\_ids part of the current partition*  leader\_partition\_scenario <- test\_case.leader\_partitions[pacemaker.current\_round - 1]   current\_partition <- []  dropped\_messages <- []   for partition in leader\_partition\_scenario.partitions:  network\_partition <- partition.partitions   if twin\_id in network\_partition:  current\_partition <- network\_partition  dropped\_messages <- partition.dropped\_messages  break   if message\_type in dropped\_messages:  *// Intra partition message drop*  return   for to\_validator\_twin\_id in to\_validator\_twin\_ids:  *// Send to all recipients who are in the same partition*  if to\_validator\_twin\_id in current\_partition:  to\_validator <- validator\_map[to\_validator\_twin\_id]  super().send(payload, to=to\_validator) Procedure should\_partition(message\_type, round\_num):  if round\_num > test\_case.n\_rounds or message\_type not in {'Proposal', 'Vote', 'Timeout'}:  return False   if message\_type in {'Proposal', 'Vote'}:  return True  else:  *// Else, message type is Timeout*  *// If we have not sent a timeout message for this round, return True, else False*  ans <- round\_num not in self.timed\_out\_rounds  self.timed\_out\_rounds.add(round\_num)   return ans |
| --- |