DiemBFT Documentation and Test Report

Team Name: RVP

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

Setup

Test case Template

```
{
      'Name'           : "Name of the test case"
      'faultyReplicas': "Number of faulty replicas to spawn",
      'timeoutDelta'  : "paceMaker message timeout in milliseconds",
      'clients'       : "number of clients to spawn",
      'requests'      : "number of transactions to be executed",
      'clientTimeout' : "timeout for client requests",
      'testcase'      : {
        "type"            : "a field use to indicate the replica class to be used",
        "specialArguments": "a dictionay of special arguments such as random seed, etc..."
      }
},
```

The following is the test case template used for all the test cases. In the figure shown above, descriptions of all fields are provided.

Generating Test cases

The test cases are generated using the testdiem.da file. The following code fetches the appropriate replica object.

```
def getReplicafromConfiguration(scenario):
    if scenario == "omission":
      return new(diem_replica_omission.Replica_Omission), self.specialArguments
    elif scenario == "normal":
      return new(diem_replica.Replica), {}
    elif scenario == "forge_signature":
      return new(diem_replica_forge.Replica), self.specialArguments
    elif scenario == "delay":
      return new(diem_replica_delay.Replica_Delay), self.specialArguments

def run:
  .........
  # spawning faulty replicas
  while i < self.faulty_replicas:
    replica, special = self.getReplicafromConfiguration(self.scenario)
    specialArgs[i] = special
    replicas.append(replica)
    i += 1

  # spawning normal replicas
  while i < replicas_required:
    replica, special = self.getReplicafromConfiguration("normal")
    specialArgs[i] = special
    replicas.append(replica)
    i += 1

  ............
```

The test cases are generated using the testdiem.da file. The following code fetches the appropriate replica object. The run method uses getReplicaConfiguration to use the appropriate replica based on the scenario specified. It can run multiple test cases one after another if the at-line position 202 in the code return value is an array.

The non-faulty replicas which are equivalent to 2*f + 1 are invoked using the normal scenario.

Logging files and Ledger Info

The logging file is generated based on the file specified during the command. using the command specified in the readme will generate a log in out.log file.

The ledger is stored on a flat file in the /tmp directory corresponding to diemLedger_*. (Here * corresponds to the replica/validator ID).

To obtain the ledger after the test run. You can run the `testdb.py` file.
The following is a sample output :

```
(py36) vscode → /workspaces/diemBFT/DiemBFT/src (release ✗) $ python testdb.py


 Commits for replica  0
Parent Block ID                                                   Block transaction    Block ID
----------------------------------------------------------------  -------------------  ----------------------------------------------------------------
0                                                                 0                    0
0                                                                 1-0                  65686032487e62dbd357b9e6f973523aca3e4b0736f677bfb3dc485401e8bd6d
012c26c83f31d17f4e8d1d147027813d785bab9c6eaf746a22cf5508a9f96da5  10-0                 4d8ce626ce0683c34c39af47def4a8d5bf63ea2e697856fbc8e7e84e0d1ae695
65686032487e62dbd357b9e6f973523aca3e4b0736f677bfb3dc485401e8bd6d  2-0                  a53c82a10849a921022acf56dad2baab36b39819c455b217c3c5a9dc5f2bf6b0
a53c82a10849a921022acf56dad2baab36b39819c455b217c3c5a9dc5f2bf6b0  3-0                  48ebb55cfb53cec080c8bfcf23e4ed44147618a6bf69b17640fbf2a3b983322a
48ebb55cfb53cec080c8bfcf23e4ed44147618a6bf69b17640fbf2a3b983322a  4-0                  e7c8d70689f93badad1bd820a405cb0c0a3be662f3c90d29cd617ce5b285a9d4
e7c8d70689f93badad1bd820a405cb0c0a3be662f3c90d29cd617ce5b285a9d4  5-0                  246b1098c695e5357eefd5ea2fb0644696cc60341bb99701d34bc868490da165
246b1098c695e5357eefd5ea2fb0644696cc60341bb99701d34bc868490da165  6-0                  959f5f47343c11c45efb8f80ba2ea04de24f4e5bab4c203d6b8541cfeef05826
959f5f47343c11c45efb8f80ba2ea04de24f4e5bab4c203d6b8541cfeef05826  7-0                  35c8ed996a0116350cb91fe415a033a01f00655e1b97910cb381f5f353f99bbe
35c8ed996a0116350cb91fe415a033a01f00655e1b97910cb381f5f353f99bbe  8-0                  0e2c3e9a4bd87e9c6373203079b88b1e3aacbd6eb69f75b22fe1e0947ad557b1
0e2c3e9a4bd87e9c6373203079b88b1e3aacbd6eb69f75b22fe1e0947ad557b1  9-0                  012c26c83f31d17f4e8d1d147027813d785bab9c6eaf746a22cf5508a9f96da5
```

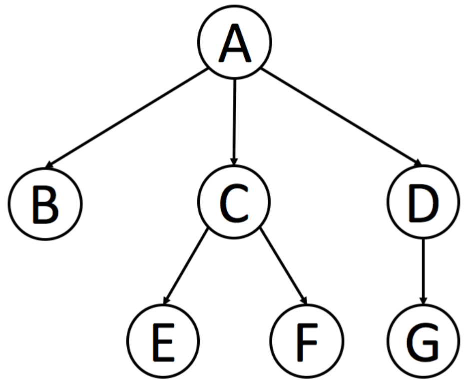
Note: After running the script, it clears the ledger. Also, the Block transaction is a string that is sorted. So although 10-0 appears first it s the last block getting committed. This can be verified by looking at the parent block ID.

Implementation

(This section is just to provide detailed explanation about our design decisions and reasoning. For actual pseudo-codes refer to the pseudo code document).

Pending Block Tree

The pending block tree is essentially a trie-like / n-ary tree data structure with the ability to perform pruning.



In the figure shown above A, B, C, etc are essentially block-ids. Each node contains a dictionary/map for its children. For example, A will contain B, C, D as its children, likewise C will contain E, F and D will contain G.

For making the lookup / adding / pruning transactions quick we have used a cache on top of this. The cache has a pointer to the reference in the tree.

The following is the pseudo-code for adding a transaction into the tree.

```
def add(prev_node_id, block):
        node =  self.get_node(prev_node_id)
        if node == null:
            node = root

        node.childNodes[block.id]=Node(prev_node_id,block)
        cache[block.id]=node.childNodes[block.id]
```

The following is the pseudo-code for pruning a node

```
def prune(self,id):
  curr_node =  self.get_node(id)

  if curr_node == null:
      return
  self.root =  curr_node
  self.cache_cleanup(id)

def cache_cleanup(id):
  cache = {}
  prune_helper(root)

def prune_helper(node):
    if node is None:
        return

    cache[node.block.id]=node

    for block_id in node.childNodes.keys():
        cache[block_id] = node.childNodes[block_id]
        prune_helper(node.childNodes[block_id])
```

MemPool

The mempool uses a queue for holding the incoming transactions. It also uses a dictionary known as a locator for storing the client until the transaction is committed. If in case a transaction is chosen by a replica for proposal it uses a set (state) for indicating the transaction is in use.
The state and locator together are used for preventing request-duplication whenever client timeout is reached a new request is given.

```
Module MemPool:
    init():
        self.queue = queue()
        self.locator = {}
        self.state = set()
```

Pseudo-code for getting a transaction

```python
def get_transactions():
        # currently only sends one transaction
        if queue has elements:
            command = queue.deque()
            # command is present in locator and command not currently processing
            if command in locator and command not in state:
                self.state.add(command)
                return command
            else:
                return self.get_transactions()
        else:
            return None
```

Pseudo-code for inserting a command

```python
def insert_command(command, client):
        # command not present in locator and command is not processing
        if command not in self.locator and command not in self.state:
            self.queue.append(command)
            self.locator[command] = client
        else:
            print("Command already present in mempool")
```

Persistent Ledger

For ledger persistent, each replica is storing its ledger in a flat-file inside the temp directory. The API used for storing and retrieving blocks is using LevelDB. For storage, we are using block_id as the key and block as the value.

While writing we make use of the sync flag which ensures flush happens before execution is returned back to the validator.

Only the committed blocks are stored inside the persistent ledger.

Speculative Ledger

The speculative ledger is similar to the persistent ledger. The only difference is that it stores the pending blocks as well. Thus in some way, it becomes storage of block to the block ids in the pending block tree.
Instead of keeping the speculative ledger in memory, we are using a flat file similar to a persistent ledger. This ensures the process does not run out of memory.

Initiation of Diem Process and chain processing

The initiation happens with round -1. Leader election chooses replica 0 as the first leader just for this round and advances_round_qc. Afterwhich block 1 is proposed and a new QC gets generated after receiving required vote messages. After generating a new QC, replica 0 again makes a proposal, and the next leader, replica 1 waits for vote messages.

The reason why we use current_round as -1 as opposed to round 0 as mentioned in the paper is to start with an even length cycle. If we start with 0, the vote messages for the proposal will be sent to replica 1 which is the next leader. (round 0 -> advance round to 1 -> broadcast proposal message -> send vote to next leader (current round 1 + 1) / 2 which will be replica 1).

Genesis Block

The genesis block is the very first block in the ledger. It contains 0 as its block ID and references parent ID as itself. It is portrayed as being formed at round -1, likewise, its parent round is portrayed as -1 as well.

Similar to the genesis block, genesis QC is formed as well. Which contains no votes as signatures but has been authored as 0.

During leader election, there are conditions in place such that the genesis block or genesis QC is not used for electing reputation leaders.

Client Request

The client request object holds the transaction as payload as well as the signature of the payload.
When the transaction is processed the client can request a replica to provide the committed block from the ledger and then use the payload to verify the signature with the original payload.

```python
class ClientRequest:
    def init(payload, source, pvt_key):
        payload = payload
        source = source
        signature = sign(payload, pvt_key)
```

Replica Info

The replica info object is used to provide the metadata about a replica to other replicas/clients. It contains the public key of the replica, the process ID used to send messages, and its replica ID.

```
class ReplicaInfo:
    def init(process, public_key, replicaID):
        process = process
        public_key = public_key
        replicaID = replicaID
```

Chain Termination

Diem follows a 2-chain protocol where the Block proposed at round N will be committed at round N + 2 at all replicas. In order to ensure the termination of all transactions sent by the client, we make use of 2 additional dummy blocks. These dummy (empty) blocks serve as placeholders so that whatever client transaction has been sent gets committed. The only flaw with this approach is that the first dummy block will get committed at any one of the replicas. As QC for dummy block, 2 gets formed but not propagated as there are no further transactions in mempool.

As these dummy blocks are empty their presence has no impact on the state of the blockchain. For example: In a monetary blockchain, these blocks can be replaced with 0 value denominations. So a transaction like that will have no effect on the final state.

Another way of providing this functionality is to generate blocks whenever get_transactions is called. (This is the way the real-world Blockchain system maintains liveness). Another way is to generate the blocks and see if the last two generated blocks are dummy or empty blocks and terminate based on that behaviour.

In case no transactions are available for processing, the replicas go into an await where they wait for a transaction to appear to continue processing again. Round numbers will get incremented. This can be further optimized by not setting a timer for a round when the mempool is empty.

Test Case Report

All configuration files are in testdiem.da. Logging is common as specified by the run command. Check test setup section above for more details.

Normal Execution Flow

```
{
        'Name'           : "Normal Replicas",
        'faultyReplicas': 1,                # number of replicas which can go faulty
        'timeoutDelta'  : 2500,             # milliseconds
        'clients'       : 1,                # number of clients to spawn
        'requests'      : 10,               # number of requests
        'clientTimeout' : 5,
        'testcase'      : {
          "type"             : "normal",
          "specialArguments": {}
        }
}
```

The format is as defined in the test template above. This case does not have any byzantine behavior. The log file is as defined in the test setup above.
The expected output is all 10 transactions to be executed and committed into the ledger. The resulting ledger state should look similar to the image below.

```
(py36) vscode → /workspaces/diemBFT/DiemBFT/src (release) $ python testdb.py

 Commits for replica  0
Parent Block ID                                                   Block transaction    Block ID
----------------------------------------------------------------  -------------------  ----------------------------------------------------------------
0                                                                 0                    0
0                                                                 1-0                  8d5437578205fa579154fb7e8f89141963dd615c92650f3d03f33a4182ef5bca
2cd769adb0e06ae677b217d5ebb81c948a9a0dc6134cf0da530dc0b0ffaba7e0  10-0                 81c84bf38bf75d1d622fa3e9fed57437661d6c2455edd267169c89379ebdcecb
8d5437578205fa579154fb7e8f89141963dd615c92650f3d03f33a4182ef5bca  2-0                  8f62a570866cfbed4224a72a79c43101d94e4ab399293ecaf206cf5b56f37830
8f62a570866cfbed4224a72a79c43101d94e4ab399293ecaf206cf5b56f37830  3-0                  1d9ae1369aec6ec88947e5294a66fa31a987916623147d7be56d1e298f6536db
1d9ae1369aec6ec88947e5294a66fa31a987916623147d7be56d1e298f6536db  4-0                  97c48114e9300a08c77ebe469595ef1ac853f1c77b2176d89c35f626e3d006c9
97c48114e9300a08c77ebe469595ef1ac853f1c77b2176d89c35f626e3d006c9  5-0                  2cdf35fc6461f166101a12f3b67b2c39019461923af923e211597f17bb8ff444
2cdf35fc6461f166101a12f3b67b2c39019461923af923e211597f17bb8ff444  6-0                  164bc02b46bfccbc70fadc5d7eeed2368496951d311f131a58ee4b49b732db60
164bc02b46bfccbc70fadc5d7eeed2368496951d311f131a58ee4b49b732db60  7-0                  5b9254a5beefc21b06b6fc72170a61ec6fb2d53b190a20e06f57b1e92f00d4f4
5b9254a5beefc21b06b6fc72170a61ec6fb2d53b190a20e06f57b1e92f00d4f4  8-0                  f282a9d8218514c4c8f8a9658f4fbeac50cdb74fffaf5ec667dbdadd6f0a582a
f282a9d8218514c4c8f8a9658f4fbeac50cdb74fffaf5ec667dbdadd6f0a582a  9-0                  2cd769adb0e06ae677b217d5ebb81c948a9a0dc6134cf0da530dc0b0ffaba7e0
```

Normal Replicas with multiple clients

```
{
      'Name'           : "Normal Replicas with multiple clients",
      'faultyReplicas': 1,
      'timeoutDelta'  : 2500,
      'clients'       : 2,
      'requests'      : 10,
      'clientTimeout' : 5,
      'testcase'      : {
        "type"              : "normal",
        "specialArguments": {}
      }
    },
```

This is similar to normal execution flow, instead, here there are two clients which are provided.

```
(py36) vscode ➜ /workspaces/diemBFT/DiemBFT/src (release ✗) $ python testdb.py


 Commits for replica  0
Parent Block ID                                                   Block transaction     Block ID
----------------------------------------------------------------  --------------------  ----------------------------------------------------------------
0                                                                 0                     0
0                                                                 1-0                   abfe654d2af449d71f83d2d113109e69ea2d34de22705e9ac18c291208c3ba86
abfe654d2af449d71f83d2d113109e69ea2d34de22705e9ac18c291208c3ba86  1-1                   134bdc7978baa76191e550dda2ea6b45f035a3b6fdfaecbadcbed147a8439773
3315f2ce128418ce538399e828bfb50fc9174e484661193aa0e3aa05f91e7b15  10-0                  57c849d5be42242d8c711b5efabd96590ee78ecef6a8df189715a6cd29c0a235
79af4ffdea0a7140d6bd250f3fdd88fae219efaaae70475d266d4caa82647f8b  10-1                  8a0f444b00a66ae453e7b9e034175476b9fee5a021b4b84cc07f9a6a210a9007
d839b90ef624bcd70a5b8bc293e6bf7c9afd1490a44b65e895285501e6615a1f  2-0                   1b26d6317d23284476251b0d97f98614b88cf41a9360b5d8897c7063bbec6252
134bdc7978baa76191e550dda2ea6b45f035a3b6fdfaecbadcbed147a8439773  2-1                   d839b90ef624bcd70a5b8bc293e6bf7c9afd1490a44b65e895285501e6615a1f
b6cda38a0aad2f8f1a8a28a37384bf6cee242db625971fb414d358a903dd4c75  3-0                   fab77da5879ae4479e0e4fc11470da8068c3b433d8f54ed754959c88904db347
1b26d6317d23284476251b0d97f98614b88cf41a9360b5d8897c7063bbec6252  3-1                   b6cda38a0aad2f8f1a8a28a37384bf6cee242db625971fb414d358a903dd4c75
4ed501340dadeea8694d50a8d652ecaf10b8606e2ecef207611b0509cd2806d5  4-0                   d9b346099149c6763d718782eb17b57d4ca32f49cec985eb2e92876c47a89381
fab77da5879ae4479e0e4fc11470da8068c3b433d8f54ed754959c88904db347  4-1                   4ed501340dadeea8694d50a8d652ecaf10b8606e2ecef207611b0509cd2806d5
85ad58b0973f5d794e73882417c6920cb207a11597850b3e6fd7960955940ffc  5-0                   0582ee4be28c72ad485ed27548c35782cb3118edf3b4229f1b8c615e492643dd
d9b346099149c6763d718782eb17b57d4ca32f49cec985eb2e92876c47a89381  5-1                   85ad58b0973f5d794e73882417c6920cb207a11597850b3e6fd7960955940ffc
0582ee4be28c72ad485ed27548c35782cb3118edf3b4229f1b8c615e492643dd  6-0                   099598db92f719badec7fe8462520b9f004c87c0c1251e15f67bd170d108e0fc
099598db92f719badec7fe8462520b9f004c87c0c1251e15f67bd170d108e0fc  6-1                   175c90d4fc17b13209199faa4ba5816adca2a769d054302060d36f4fcbc20c46
175c90d4fc17b13209199faa4ba5816adca2a769d054302060d36f4fcbc20c46  7-0                   dcf3c7165e7b80014bf2e1c03c057f24b7a9d3e2efb085f1035ead6220197bdb
dcf3c7165e7b80014bf2e1c03c057f24b7a9d3e2efb085f1035ead6220197bdb  7-1                   f40a467f51e6a01d0afc53136fd6419892d571f8dc305ae74c07e00de94cfa75
f40a467f51e6a01d0afc53136fd6419892d571f8dc305ae74c07e00de94cfa75  8-0                   0d39dbd170326102c567c6fd14eb88b10eadf41a392e91b8ed75367026b78980
8f459ecfd8fc0d6ae1eb6c88d7f97ff92bb414618b883d8bff00ee93056a99e4  8-1                   3315f2ce128418ce538399e828bfb50fc9174e484661193aa0e3aa05f91e7b15
0d39dbd170326102c567c6fd14eb88b10eadf41a392e91b8ed75367026b78980  9-0                   8f459ecfd8fc0d6ae1eb6c88d7f97ff92bb414618b883d8bff00ee93056a99e4
c709d09a0f619abd020ae2d5d88a7f2673ed249f2df4553cc0605e0074af29ce  9-1                   79af4ffdea0a7140d6bd250f3fdd88fae219efaaae70475d266d4caa82647f8b
57c849d5be42242d8c711b5efabd96590ee78ecef6a8df189715a6cd29c0a235  dummy1-0              c709d09a0f619abd020ae2d5d88a7f2673ed249f2df4553cc0605e0074af29ce
```

Client requests timeout, re-submission, and handling request de-duplication

```
{
      'Name'          : "Client small timeout with request resubmission and handling de-deuplication",
      'faultyReplicas': 1,
      'timeoutDelta'  : 2500,
      'clients'       : 1,
      'requests'      : 10,
      'clientTimeout' : 0.5,
      'testcase'      : {
        "type"             : "normal",
        "specialArguments": {}
      }
    },
```

In this test case we set the client timeout to be very low i.e 500ms due to which, the client keeps resending the request on timeout. On receiving a duplicate client request the replicas should handle them and only commit the unique ones.

```
(py36) vscode ➜ /workspaces/diemBFT/DiemBFT/src (release ✗) $ python testdb.py


 Commits for replica  0
Parent Block ID                                                    Block transaction    Block ID
----------------------------------------------------------------   ------------------   ----------------------------------------------------------------
0                                                                  0                    0
0                                                                  1-0                  5334759456abea3d021606c7f8fef2566643ed3fbec777bf8a2068997971fc93
92e8d6073ee32142c9f3a6c9031614cdab4d5c058d1fe2eb9450108573e72b20   10-0                 8c8e4b202fc9e69761d63d045418eca674743fa2d5790824a8dc4d6b28d388e9
5334759456abea3d021606c7f8fef2566643ed3fbec777bf8a2068997971fc93   2-0                  4fc1c03add94a587b408c8bad051827fb081ce7db666566aa0c3976617e6d3d7
4fc1c03add94a587b408c8bad051827fb081ce7db666566aa0c3976617e6d3d7   3-0                  33f12e3c1040c0c47b111af992bb9424114f097928a8c3822de877d4d1916407
33f12e3c1040c0c47b111af992bb9424114f097928a8c3822de877d4d1916407   4-0                  1d3781f41c8798ba1ffbfa5ac1e47cb46742957ef0d64746e7a2aae205153bda
1d3781f41c8798ba1ffbfa5ac1e47cb46742957ef0d64746e7a2aae205153bda   5-0                  f5f728ae0fd7c9372884d26946458359fbb6217ed4ff526c05b2e29b2be49cc3
f5f728ae0fd7c9372884d26946458359fbb6217ed4ff526c05b2e29b2be49cc3   6-0                  e9ed28f5dff94ca114d9e6082360bd78fba1bff734d183cecb28674161b16ee3
e9ed28f5dff94ca114d9e6082360bd78fba1bff734d183cecb28674161b16ee3   7-0                  f4d8271f0b16c765a0566609ef291465e575439a08ee6fa98e72ca9ebb65e48c
f4d8271f0b16c765a0566609ef291465e575439a08ee6fa98e72ca9ebb65e48c   8-0                  c98a1a9a103515c87fff8cca4df47e232e9c1132ce68ce4723cfba76fe3f667f
c98a1a9a103515c87fff8cca4df47e232e9c1132ce68ce4723cfba76fe3f667f   9-0                  92e8d6073ee32142c9f3a6c9031614cdab4d5c058d1fe2eb9450108573e72b20
```

Omission Failure

```
{
      'Name'           : "Faulty replica having omission failures",
      'faultyReplicas': 1,
      'timeoutDelta'  : 2500,
      'clients'       : 1,
      'requests'      : 5,
      'clientTimeout' : 5,
      'testcase'      : {
        "type"            : "omission",
        "specialArguments": {}
      },
    }
```

In the following test case the faulty replicas cause omission failures but not providing a Vote Message. This indicates the tolerance of DiemBFT to delays in the network.

```
(py36) vscode → /workspaces/diemBFT/DiemBFT/src (release ✗) $ python testdb.py

 Commits for replica  0
Parent Block ID                                                   Block transaction    Block ID
----------------------------------------------------------------  -------------------  ----------------------------------------------------------------
0                                                                 0                    0
0                                                                 1-0                  dd131503e2a6ba571d70abccba82f67ea49663269f9f6c1924907256fe3ffef1
dd131503e2a6ba571d70abccba82f67ea49663269f9f6c1924907256fe3ffef1  2-0                  df5091e16d6f062d5c1df9cebb72bc61e1317c30cf8c854fc72e3ebc12134db0
df5091e16d6f062d5c1df9cebb72bc61e1317c30cf8c854fc72e3ebc12134db0  3-0                  a52e27aea4a73b35ef6b35e0958be4aa08827db59aae0814a12ec81a1cecf7d9
a52e27aea4a73b35ef6b35e0958be4aa08827db59aae0814a12ec81a1cecf7d9  4-0                  b38190ae961321774b2098e41cb695f1fb4ec3d3e41dd58de0b35e3d29039068
b38190ae961321774b2098e41cb695f1fb4ec3d3e41dd58de0b35e3d29039068  5-0                  79e709099801de37be024c002afde54b56566230acc517cd4fb8c17e06211c3d
```

Forge Signature

In the following test case, the faulty validator tries to forge the QC signature. Which causes a timeout and later leads the chain to recover. In our case, due to the design of mempool, new requests sent by clients do not get added to mempool as they were trying to be processed earlier. This is further discussed in bugs and limitations. (Note: the process does not terminate by itself as the client has not received all acks for all requests which were requested). In the figure, the client does try to re-transmit request 1, 2 but th mempool does not insert them.

```
{
      'Name'           : "Forge signature",
      'faultyReplicas': 1,
      'timeoutDelta'  : 500,
      'clients'       : 1,
      'requests'      : 5,
      'clientTimeout' : 2,
      'testcase'      : {
        "type"              : "forge_signature",
        "specialArguments": {}
      },
    },
```

```
^C(py36) vscode ➜ /workspaces/diemBFT/DiemBFT/src (release ✗) $ python testdb.py


 Commits for replica  0
Parent Block ID                                                   Block transaction    Block ID
----------------------------------------------------------------  -------------------  ----------------------------------------------------------------
0                                                                 0                    0
0                                                                 3-0                  d8c43eb86c8e22caf7f95e0eb9f27370429d36204009bcc8acaa0438d65fc0c8
d8c43eb86c8e22caf7f95e0eb9f27370429d36204009bcc8acaa0438d65fc0c8  4-0                  85b8d45c60a127d55daba6494536b69fb9002bd10723829392da8d12aec58288
85b8d45c60a127d55daba6494536b69fb9002bd10723829392da8d12aec58288  5-0                  f4e94b208eeeb220b0de331430c6c9e8ff34c925a3aa5eb64521c3f750d569df
f4e94b208eeeb220b0de331430c6c9e8ff34c925a3aa5eb64521c3f750d569df  dummy1-0             70dab0bd8ab0734c9fe907699eb5722a5826192dd8b4d54a5080c4133a391752
```

Delay Failures

```
{
        "Name"           : "Faulty replicas having delay failures",
        "faultyReplicas": 1,
        "timeoutDelta"  : 2500,
        "clients"       : 1,
        "requests"      : 5,
        "clientTimeout" : 5,
        "testcase"      : {
            "type"             : "delay",
            "specialArguments": {
                "randomSeed"                  : 50,
            }
        }
    }
```

In delay failure, we simulate network delay using a random seed to timeout a message which is to be sent. This has a similar issue to the former test case where it fails due to the design of mempool. (Note: the process does not terminate by itself as the client has not received all acks for all requests which were requested).