Q1 + Q2: 2PC Implementation:

For our implementation we used Python to write the Voting Phase of each node and Java to write the Decision phase of each node. For each phase, there are two scripts, one for coordinator and one for all participants. We define a common .proto file for both the both phases as well as coordinator and participants. The services we define in the twopc.proto file are shown below.

```
service VotingPhase {
  rpc RequestVote (VoteRequest) returns (VoteResponse);
}
service DecisionPhase {
  rpc GlobalDecision (DecisionRequest) returns (DecisionResponse);
}
service DecisionCoordinatorService {
  // This RPC is invoked by the Python coordinator to hand off the decision.
  rpc startDecisionPhase (DecisionHandoffRequest) returns (DecisionHandoffResponse);
}
```

Voting Phase:

```
service VotingPhase {
  rpc RequestVote (VoteRequest) returns (VoteResponse);
}

message VoteRequest {
  string vote_request = 1;
}

message VoteResponse {
  bool vote_commit = 1;
}
```

- In the Voting Phase, the coordinator acts as a client and uses the rpc **RequestVote** to send **VoteRequest** to each participant server. And each participant replies with a **VoteResponse** *(vote_commit = True/False)* for each rpc call by the coordinator.

- If coordinator gets True for all **VoteRequest** from all participants, the coordinator decides to globally commit otherwise coordinator will globally abort.

HandOff Phase:

```
service DecisionCoordinatorService {
  // This RPC is invoked by the Python coordinator to hand off the decision.
  rpc startDecisionPhase (DecisionHandoffRequest) returns (DecisionHandoffResponse);
}

message DecisionHandoffRequest {
  // The global decision computed by the Python voting phase.
  bool global_commit = 1;
  repeated string participant_addresses = 2;
}

message DecisionHandoffResponse {
  string message = 1;
}
```

- Next starts the Decision Phase with a handoff of decision from Voting Phase to Decision Phase. Here, the *python Coordinator* will handoff the decision (global_commit = True/False) by using the rpc **startDecisionPhase** to the *java Coordinator*.

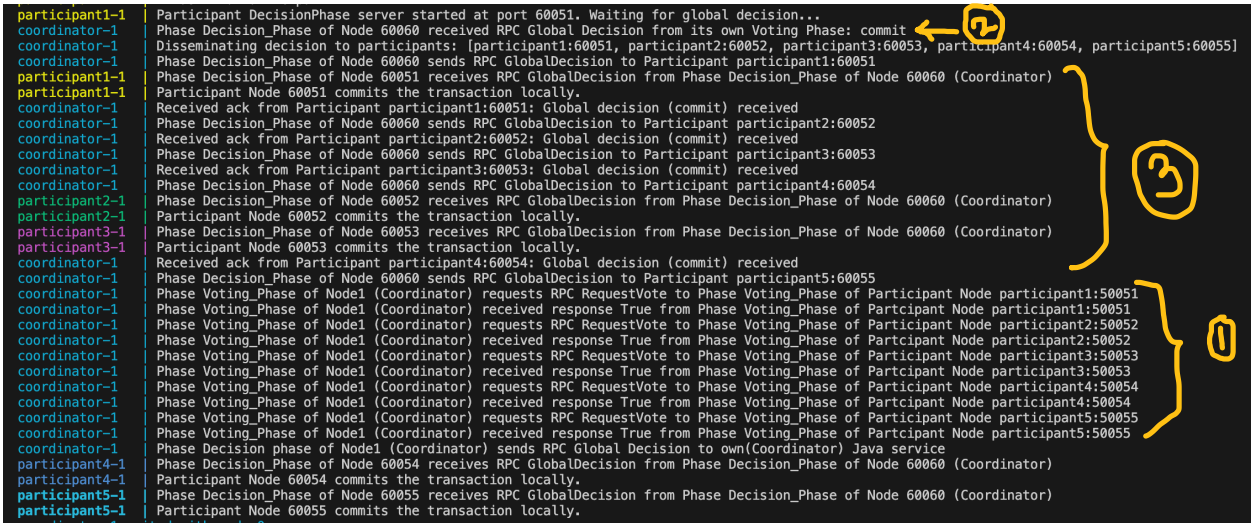
- The python coordinator will use the **DecisionHandoffRequest** to send the global_commit decision and list of all participant_addresses to the java coordinator and the java coordinator will reply with a message such as '*Global decision disseminated to all participants'*

Decision Phase:

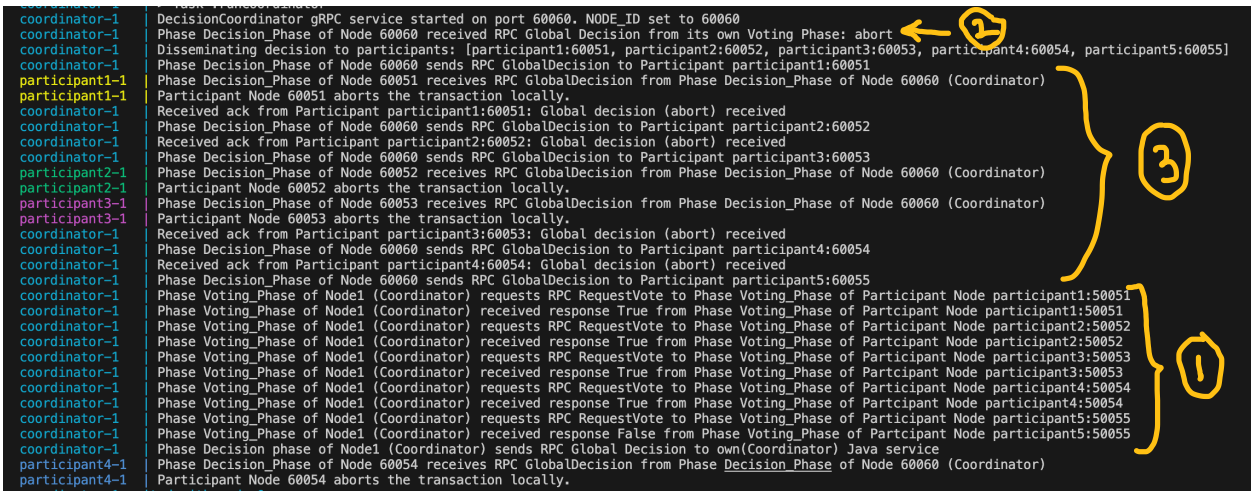
```
service DecisionPhase {
  rpc GlobalDecision (DecisionRequest) returns (DecisionResponse);
}

message DecisionRequest {
  bool global_commit = 1;
}

message DecisionResponse {
  string ack = 1;
}
```

- Once the decision has been handed off to Decision Phase (DecisionCoordinator.java), the java coordinator uses rpc **GlobalDecision** to disseminate the decision using **DecisionRequest** to all other participants. And each participant acknowledges the decision and either commits or aborts the operation locally.

Output:



In this screen shot, all the participant nodes are up and running**, Area marked with 1** shows the voting phase where coordinator sends RequestVote to each participant and gets acknowledgement from all nodes. In **area marked 2,** the Decision phase received the decision from Voting Phase which is commit. In **area marked 3**, the coordinator (Decision Phase) disseminates the Global commit decision to all participants and all participants commit the transaction locally.



In this screenshot, node 5(50055:60055) is not running, Hence in Voting Phase, **Area marked as 1**, the coordinator received True response from all nodes except node 5. Hence, the Coordinator decides to abort, and the Decision phase gets the Global Decision:

abort in **area marked 2**. Then, in **area marked 3**, the decision coordinator disseminates the global abort to all participants and thus all participants aborts the transaction locally.

Q3 + Q4: Raft Implementation:

For our implementation, we used **Python** to handle the leader election of Raft and **Java** to handle the log replication of each node. For each phase, there is only one script: **leader_election.py** and **RaftLogReplicationServer.java**. Additionally, client requests are handled by **RaftClient.java**. We define a common .proto file for both phases. The services we define in the **raft.proto** file are shown below.

```
service Raft {
  // RPC used during leader election
  rpc RequestVote(VoteRequest) returns (VoteResponse);
  // RPC used both as heartbeat and for log replication
  rpc AppendEntries(AppendEntriesRequest) returns (AppendEntriesResponse);
  // RPC used by clients to submit an operation/command
  rpc SubmitOperation(OperationRequest) returns (OperationResponse);
  // RPC used to hand off new leader_id after leader election
  rpc HandoffLeader (ChangedLeader) returns (ChangedLeaderAck)
}
```

Leader Election:

- There are two variables defined at first:
    - **`HEARTBEAT_INTERVAL = 1.0`**
    - **`ELECTION_TIMEOUT = random.uniform(1.5, 3.0)`**
- Each node will start in a *'follower'* state.
- When all the nodes start running, they will wait for an interval as specified by **`ELECTION_TIMEOUT`**. If the nodes do not get a heartbeat within that interval, each node will start an election according to their own specified interval and move to the *'candidate'* phase.

```
message VoteRequest {
  int32 term = 1;
  string candidateId = 2;
}

message VoteResponse {
  int32 term = 1;
  bool voteGranted = 2;
}
```

- When a particular node moves to the candidate phase, it increments the **`CURRENT_TERM`** (election term) and acts as a client to send **`VoteRequest`** through the **`rpc RequestVote`** to collect votes from all other members.

- When the **`RequestVote`** rpc of other member nodes is invoked, the node will first check for two conditions:
    - If the election term of the requesting node is more updated than their current term
    - If this node has not already voted for anyone else in the current term

- If both conditions are satisfied, the particular member node will grant the vote request and send a positive response via **`VoteResponse`** of **`RequestVote rpc`**. Otherwise, it will not grant the **`VoteRequest`**.

- After receiving the votes, the candidate node will check if:
    - **VOTES_RECEIVED > len(MEMBER_NODES) /2**

- If the condition is True, it will move to the *'leader'* state, otherwise it will revert back to the *'follower'* state.

```
message LogEntry {
  string operation = 1;
  int32 term = 2;
  int32 index = 3;
}

message AppendEntriesRequest {
  int32 term = 1;
  string leaderId = 2;
  // The full log list from the leader
  repeated LogEntry log = 3;
  // index of the most recently committed operation
  int32 commitIndex = 4;
}

message AppendEntriesResponse {
  int32 term = 1;
  bool success = 2;
}
```

- When a candidate node becomes the *leader*, it invokes the **`AppendEntries`** RPC to send initial logs to all member nodes (for synchronization purposes only). The actual log replication is handled by the Java end.

- When member nodes receive the **`AppendEntriesRequest`** from the leader node, any node in the candidate phase attempting to become a leader will revert back to the *follower* state and reply with an **`AppendEntriesResponse`**.

Handoff New Leader:

```
message ChangedLeader {
  string newleader = 1;
}

message ChangedLeaderAck{
  bool ack = 1;
}
```

Once a candidate node becomes the leader, it initiates log replication through the **`HandoffLeader`**. The elected leader at the Python end uses the **`rpc HandoffLeader`** to inform its own Java server about the leader update using the **`ChangedLeader`** message.

Log Replication:

After the leader has been handed off to the Java server, the Java end of the leader node starts the log replication using **`rpc AppendEntries.`** In each **`AppendEntriesRequest`**, the leader node sends logs that contain

<operation, election_term, index of the operation in the log>

```
message LogEntry {
  string operation = 1;
  int32 term = 2;
  int32 index = 3;
}

message AppendEntriesRequest {
  int32 term = 1;
  string leaderId = 2;
  // The full log list from the leader
  repeated LogEntry log = 3;
  // index of the most recently committed operation
  int32 commitIndex = 4;
}

message AppendEntriesResponse {
  int32 term = 1;
  bool success = 2;
}
```

Client Request:

```
message OperationRequest {
  string operation = 1;
}

message OperationResponse {
  string result = 1;
}
```

When the client submits a request using the **`rpc SubmitOperation`**, there are two cases:

Case 1: Client Submits Request to the Leader Node

If the client submits a request to the leader node itself (i.e., invokes the **`submitOperation`** RPC of the leader), the leader:

- Leader node appends the operation, current election_term and the index of the operation to the log.

- Sends the log (along with the current term and the most recently committed operation, not the one just invoked) using the **`AppendEntriesRequest`** of **`AppendEntries`** rpc to all member nodes (follower)
- returns an acknowledgement through the AppendEntriesResponse.
- Follower nodes return acknowledgments through **`AppendEntriesResponse`**

Note: The updated log is not sent immediately; it is sent with the next heartbeat.

Then the leader check for the condition:

ackCount > memberNodes.size() / 2

If the condition is satisfied, the leader:

- Commits the operation requested by the client
- Sends an acknowledgment to the client
- Updates **`commitIndex`** to the newly committed operation
- The updated variables is not sent immediately, it is sent with the next heartbeat.

Case 2: Client Submits Request to a Follower Node

If the client submits a request to a follower node, the follower forwards the request to the leader by invoking the **`submitOperation`** RPC of the leader. This logic is handled in **`ForwardRequest.java`**. Once the leader receives the forwarded request, it starts the log replication process as described in Case 1.

Q6: Test Cases for Raft Implementation

Test Case 1: When all nodes start as followers, the nodes start election after election_timeout and choose a leader.

- In the region marked as **1** all the nodes have started and are in the follower state. The java server (responsible for log-replication) and the python server (responsible for leader election) for each node all start in the follower state.
- In the region marked as **2** node 5053 has started the leader election process (due to the absence of heartbeats from a leader node for **`ELECTION_TIMEOUT`** seconds) on term 1.
- In region **3** we see that node 5053 gets voteGranted: True from all the other nodes as is elected the leader for term 1.
- The python service of node 5053 lets its own java service (running on 60053) know that it has been elected as the leader using the **rpc HandofLeader** and the java service of the leader node begins the LogReplication process.
- The python service of node 5053 also keeps on sending heartbeats to the other nodes so that all the other follower nodes get a heartbeat within **`ELECTION_TIMEOUT`** seconds.

Test case 2: If the leader node stops working, other member nodes hold an election and chooses a new leader from the up and running nodes.

```
raft-node-3  | Java Node 60053 sends RPC AppendEntries to raft-node-1:60051
raft-node-3  | Java Node 60053 sends RPC AppendEntries to raft-node-2:60052
raft-node-3  | Java Node 60053 sends RPC AppendEntries to raft-node-4:60054
raft-node-3  | Java Node 60053 sends RPC AppendEntries to raft-node-5:60055
raft-node-3  | Python Node 50053 sends RPC AppendEntries to raft-node-1:50051
raft-node-1  | Python Node 50051 runs RPC AppendEntries called by Leader Node 50053
raft-node-1  | Term 3
raft-node-3  | Python Node 50053 sends RPC AppendEntries to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC AppendEntries called by Leader Node 50053
raft-node-2  | Term 3
raft-node-3  | Python Node 50053 sends RPC AppendEntries to raft-node-4:50054
raft-node-4  | Python Node 50054 runs RPC AppendEntries called by Leader Node 50053
raft-node-4  | Term 3
raft-node-3  | Python Node 50053 sends RPC AppendEntries to raft-node-5:50055
raft-node-5  | Python Node 50055 runs RPC AppendEntries called by Leader Node 50053
raft-node-5  | Term 3
raft-node-3 exited with code 137
```

We see that initially node 3 is the leader and we turn it off the node from docker desk to simulate a leader node crashing.

```
raft-node-3 exited with code 137
raft-node-1  | Python Node 50051 starting election for term 4
raft-node-1  | Python Node 50051 sends RPC RequestVote to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC RequestVote called by Node 50051
raft-node-1  | term: 4
raft-node-1  | voteGranted: true
raft-node-1  |
raft-node-1  | Python Node 50051 sends RPC RequestVote to raft-node-3:50053
raft-node-1  | Python Node 50051 sends RPC RequestVote to raft-node-4:50054
raft-node-4  | Python Node 50054 runs RPC RequestVote called by Node 50051
raft-node-1  | term: 4
raft-node-1  | voteGranted: true
raft-node-1  |
raft-node-1  | Python Node 50051 sends RPC RequestVote to raft-node-5:50055
raft-node-5  | Python Node 50055 starting election for term 4
raft-node-5  | Python Node 50055 sends RPC RequestVote to raft-node-1:50051
raft-node-5  | Python Node 50055 runs RPC RequestVote called by Node 50051
raft-node-1  | term: 4
raft-node-1  |
raft-node-1  | Received 3 votes
raft-node-1  | Python Node 50051 becomes leader for term 4
raft-node-1  | Python Node 50051 contacting Java server at raft-node-1:60051 for handoff
```

When node 3 crashes on term 3 node 1 initiates an election after incrementing the term from 3 to 4. Node 3 receives a majority vote and becomes the leader for term 4.

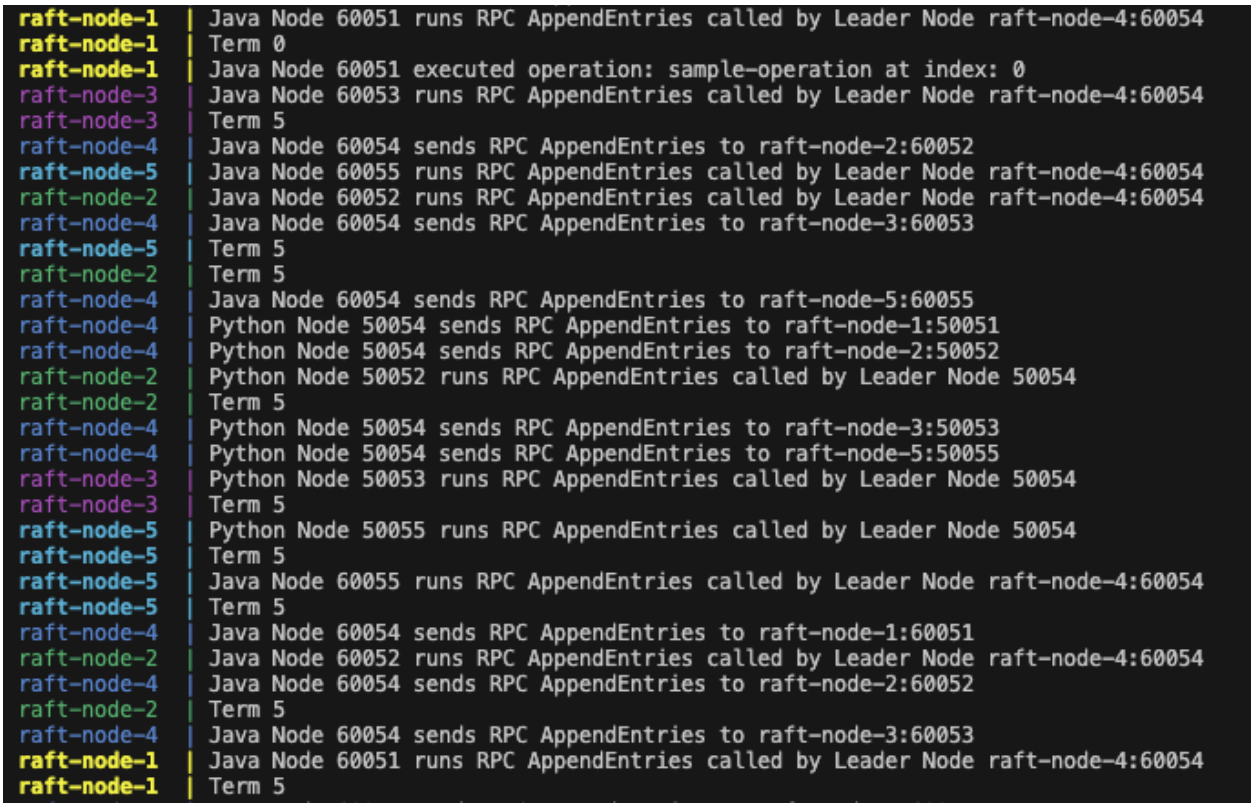
Test Case 3: If a dropped node re-joins the system, it receives the updated log and immediately synchronizes with the other nodes, starting from the current election term that the existing nodes are in.

```
raft-node-5  | Term 3
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-3:60053
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-4:60054
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-5:60055
raft-node-5  | Python Node 50055 runs RPC AppendEntries called by Leader Node 50051
raft-node-5  | Term 3
raft-node-1 exited with code 137
raft-node-5  | Python Node 50055 starting election for term 4
raft-node-5  | Python Node 50055 sends RPC RequestVote to raft-node-1:50051
raft-node-5  | Python Node 50055 sends RPC RequestVote to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC RequestVote called by Node 50055
raft-node-5  | term: 4
raft-node-5  | voteGranted: true
raft-node-5  |
raft-node-5  | Python Node 50055 sends RPC RequestVote to raft-node-3:50053
raft-node-3  | Python Node 50053 runs RPC RequestVote called by Node 50055
raft-node-5  | term: 4
raft-node-5  | voteGranted: true
raft-node-5  |
raft-node-5  | Python Node 50055 sends RPC RequestVote to raft-node-4:50054
raft-node-4  | Python Node 50054 runs RPC RequestVote called by Node 50055
raft-node-5  | term: 4
raft-node-5  | voteGranted: true
raft-node-5  |
raft-node-5  | Received 4 votes
raft-node-5  | Python Node 50055 becomes leader for term 4
```

We see that node 1 is the initial leader for term 3 and after we turn it off from docker desktop. Node 5 increments term from 3 to 4 and initiates an election. It does get a majority vote and gets elected as the leader for term 4.

```
raft-node-5  | Term 5
raft-node-4  | Java Node 60054 sends RPC AppendEntries to raft-node-5:60055
raft-node-3  | Java Node 60053 runs RPC AppendEntries called by Leader Node raft-node-4:60054
raft-node-3  | Term 5
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-1:50051
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC AppendEntries called by Leader Node 50054
raft-node-2  | Term 5
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-3:50053
raft-node-3  | Python Node 50053 runs RPC AppendEntries called by Leader Node 50054
raft-node-3  | Term 5
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-5:50055
raft-node-5  | Python Node 50055 runs RPC AppendEntries called by Leader Node 50054
```

Eventually node 4 becomes the new leader on term 5 and sends everyone heartbeats.



When node 1 rejoins the system (we turn it back on from docker desktop) it initially has its term set to 0. However, when it receives the next heartbeat it replicates the log from the leader. And we see that node 1 moves from term 0 to term 5 due to replicating the log from the heartbeat of the leader.

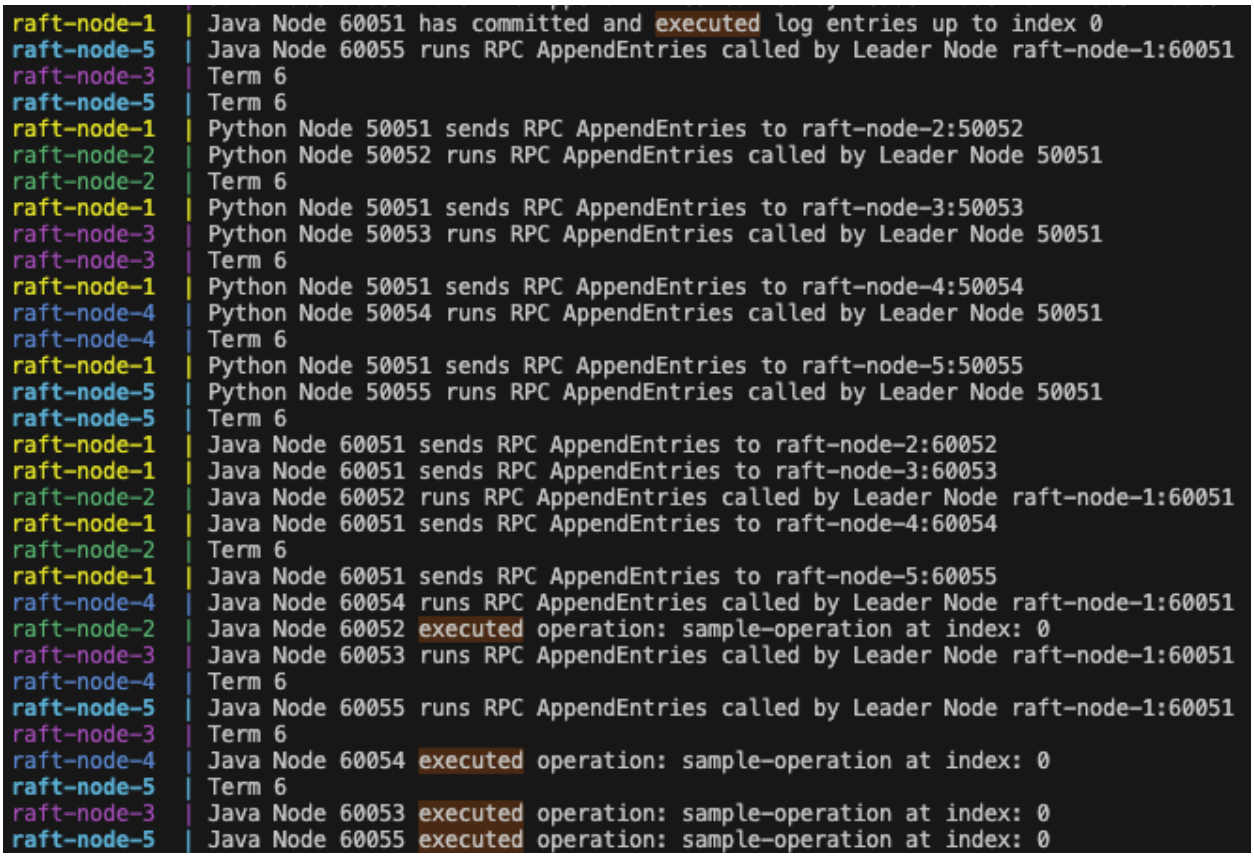
Test Case 4: When Client submits a request to the leader node, the leader node updates log and sends updated log with the next heartbeat

```
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-2:60052
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-3:60053
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-4:60054
raft-node-1  | Java Node 60051 sends RPC AppendEntries to raft-node-5:60055
raft-node-3  | Java Node 60053 runs RPC AppendEntries called by Leader Node raft-node-1:60051
raft-node-3  | Term 5
raft-node-4  | Java Node 60054 runs RPC AppendEntries called by Leader Node raft-node-1:60051
raft-node-4  | Term 5
raft-node-5  | Java Node 60055 runs RPC AppendEntries called by Leader Node raft-node-1:60051
raft-node-5  | Term 5
raft-node-1  | Python Node 50051 sends RPC AppendEntries to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC AppendEntries called by Leader Node 50051
raft-node-2  | Term 6
raft-node-1  | Python Node 50051 sends RPC AppendEntries to raft-node-3:50053
raft-node-3  | Python Node 50053 runs RPC AppendEntries called by Leader Node 50051
raft-node-3  | Term 6
raft-node-1  | Python Node 50051 sends RPC AppendEntries to raft-node-4:50054
raft-node-4  | Python Node 50054 runs RPC AppendEntries called by Leader Node 50051
raft-node-4  | Term 6
raft-node-1  | Python Node 50051 sends RPC AppendEntries to raft-node-5:50055
raft-node-5  | Python Node 50055 runs RPC AppendEntries called by Leader Node 50051
```

Here we see that the node 1 is the leader on Term 6. It is sending heartbeats to all the other nodes using its python service (running on 50051) and the log replication is done by node 1 using the java LogReplicationService (running on 60051).

```
raft-node-1  | Java Node 60051 runs RPC SubmitOperation called by Client
raft-node-1  | Java Node 60051 appended log entry: operation: "sample-operation"
raft-node-1  | term: 6
raft-node-1  |
raft-client  |
raft-client  | > Task :runClient
raft-client  | Client sends RPC SubmitOperation to raft-node-1:60051
raft-client  | Response: Operation appended to log. It will be replicated on the next heartbeat.
```

A client submits an operation to node 1 (which happens to be the leader node) using the **rpc SubmitOperation**. The leader node appends the “sample-operation” to its logs and responds to the client that it will be executed and replicated in the next heartbeat.



After node 1 (the leader node) receives acknowledgement from more than half of the follower nodes it executes the client operation. The other nodes also execute the operation in the next heartbeat when they see that the log from the leader contains a newly committed operation.

Test Case 5: When Client submits a request to a follower node, the follower node forwards the request to the leader node.

```
raft-node-4  | Java Node 60054 sends RPC AppendEntries to raft-node-2:60052
raft-node-2  | Java Node 60052 runs RPC AppendEntries called by Leader Node raft-node-4:60054
raft-node-4  | Java Node 60054 sends RPC AppendEntries to raft-node-3:60053
raft-node-1  | Java Node 60051 runs RPC AppendEntries called by Leader Node raft-node-4:60054
raft-node-5  | Java Node 60055 runs RPC AppendEntries called by Leader Node raft-node-4:60054
raft-node-2  | Term 1
raft-node-4  | Java Node 60054 sends RPC AppendEntries to raft-node-5:60055
raft-node-1  | Term 1
raft-node-5  | Term 1
raft-node-3  | Java Node 60053 runs RPC AppendEntries called by Leader Node raft-node-4:60054
raft-node-3  | Term 1
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-1:50051
raft-node-1  | Python Node 50051 runs RPC AppendEntries called by Leader Node 50054
raft-node-1  | Term 1
raft-node-4  | Python Node 50054 sends RPC AppendEntries to raft-node-2:50052
raft-node-2  | Python Node 50052 runs RPC AppendEntries called by Leader Node 50054
```

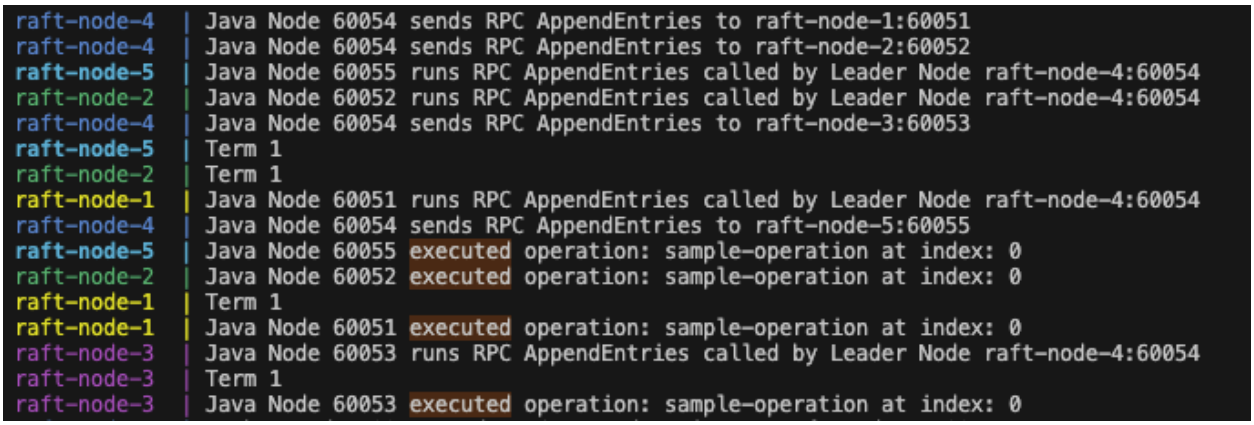
Initially node 4 is the leader and is responsible for sending heartbeats to all the other followers.

```
raft-client  | > Task :extractProto UP-TO-DATE
raft-client  | > Task :processResources UP-TO-DATE
raft-client  | > Task :extractIncludeProto UP-TO-DATE
raft-client  | > Task :generateProto UP-TO-DATE
raft-client  | > Task :compileJava UP-TO-DATE
raft-client  | > Task :classes UP-TO-DATE
raft-node-1  | Java Node 60051 runs RPC SubmitOperation called by Client
raft-node-1  | Java Node 60051 forwards RPC SubmitOperation to Leader Node raft-node-4:60054
raft-node-1  | Forwarding: LeaderPort: raft-node-4:60054
raft-node-1  | Forwarding: Node raft-node-4:60054 sends RPC SubmitOperation to Leader Node raft-node-4:60054
```

When the client runs it submits an operation to node 1. However, as node 1 is not the leader we can see that node forwards this request to the leader node's (node 4) Java service running on 60054.

```
raft-node-4  | Java Node 60054 executed operation: sample-operation at index: 0
raft-node-1  | Term 1
raft-node-4  | Java Node 60054 has committed and executed log entries up to index 0
```

When node 4 receives acknowledgement from more than half of the followers it commits and executes the operation.



When all the other nodes receive their next heartbeat they see that a new operation has been committed and executed by the leader from the logs. And they themselves execute the operation.