**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 is shown below.

A screen shot of a computer program

AI-generated content may be incorrect.

**Voting Phase:**

A screen shot of a computer program

AI-generated content may be incorrect.

* 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:**

A screen shot of a computer program

AI-generated content may be incorrect.

* Next starts the Decision Phase with a handoff of decision from Voting Phase to Decision Phase. Here, the *python Coordinator* will handoff decision (global\_commit = True/False) by using the rpc **startDecisionPhase** to the *java Coordinator*.
* The python coordinator with 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:**

A screen shot of a computer program

AI-generated content may be incorrect.

* 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.

**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.

A screen shot of a computer program

AI-generated content may be incorrect.

**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.

A computer screen with text and numbers

AI-generated content may be incorrect.

* 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.

A computer screen with text and numbers

AI-generated content may be incorrect.

* 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:**

A computer screen with text on it

AI-generated content may be incorrect.

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>*

A computer screen shot of a program code

AI-generated content may be incorrect.

**Client Request:**

A computer screen with text and symbols

AI-generated content may be incorrect.

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 follower, the nodes start election after election\_timeout and chooses a leader.

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.

Test Case 3: If a new node 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.

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

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