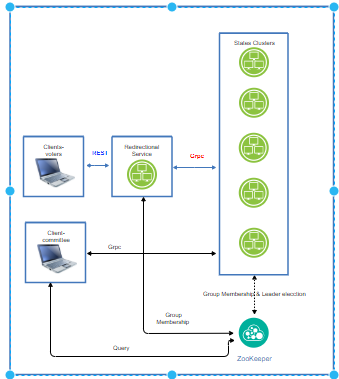
**Election in the USA – Distributed Systems Project – Documentation:**

**Our system guarantees the following:**

* **Atomicity –** If a client made a request it will either fail or succeed, there is no in-between.
* **Fault tolerance –** For each service we provide a fault tolerance environment, and each cluster can determine the amount of failures it can recover from based on the scale the service we need.
* **Linearizability –** Each state shard will have the same view of the vote count in his state.
* **Reliability –** Once a client has vote and it succeeded it will persist from that time forward, or until he overwrites his vote.
  + This is guaranteed only if the fault tolerance limit hasn’t exceeded.
* **Timeliness –** The client view on the system is guaranteed to be up to date each time he checks for the election results.

**Design**

We will start with a high-level design diagram of our voting system, and we will explain each part of the system below:



**Redirection Service:**

This service is in-charge of receiving clients requests via REST API we will elaborate on below, and it sends the clients request via Grpc to the correct state cluster depending on the voter information. It will use group membership of the state’s client and will select one of the available members from the cluster of the state.

The clients will only have direct contact with this service.

**Technical justification:**

* We want to decrease the load on the state shards, since the clients supposedly don’t know which shard they send the they request to, with this design we decrease at least 50 extra connections the state shard would have had to maintain in order to move the clients request to the correct cluster.
* We also create another level of hierarchy in the voting system and contribute to the transparency of the clients of how our systems function which adds safety elements.
* Lastly, this will help us add firewall functionality if needed in the future, which makes are system easier to expand.

**States cluster:**

Each state shard function independently, and work in leader-slave format to provide a **Total order** implementation. Meaning there is one leader per cluster which all votes must pass through so it can determine the order in which the votes have arrived, and also we make sure all the votes also register on his available slaves, and in that way we promise linearizability, and fault tolerance. In addition, we will use Zookeeper for Leader election in case a leader fails, the implementation for that will explain later, and group membership to know who are active in our state cluster.

**Technical justification:**

* We want the system to preserve Linearizability as we stated on our guarantees, we do that by having the leader select an order for all the group members to see.
* We want our system to also be fault tolerance, we do that by using the Zookeeper to implement group membership of each shard, and also leader election.
* Also, in this design we promise atomicity since we won’t ack a client request until a quorum of our cluster have submitted the result.
* We also provide reliability which is a direct result of linearizability combined with fault tolerance.
* Lastly, this design is scalable, since it’s easy to add more nodes to each state cluster if needed for being more fault tolerant.

**Note:** Communication wise this isn’t the most minimal design, but since we are dealing here only with scalability for fault tolerance and not for computing power, we can reach a very high availability rate with cluster not larger than let’s say 11 servers (

Voting process

The voter sends REST message with his vote and his state and then the redirection service redirect the vote to the right state.

The voting algorithm works as following:

* If the state’s leader got the vote request, he will send it to all the state’s slaves and will wait for ack from all of them that the vote’s update succeeded. In case all of them succeed to update the vote, the leader will also update the vote and will send to the voter succeed message through the redirection service. In case not all of the slaves succeed to update the vote, the leader will roll back the vote in all the slaves that succeeded and will send error message to the voter through the redirection service.
* If one of the state’s slaves got the vote request from the redirection service, then he will forward the vote request to the state’s leader and the leader will do the actions described in the paragraph above.
* If one of the state’s slaves receives vote request from the leader, he will try to update the vote and reply by sending back to the leader if he succeeded.
* This implementation realizes total order between the vote requests because every vote request goes through the leader that determines the order all slaves

will update the votes. In other words, all of the state’s cluster sees the same view (linearizability).

In addition, this implementation realizes atomicity by accepting vote request only if all of the cluster succeed to update it. Meaning, there is only two options: either all of the cluster’s servers accepted the vote or none of the cluster’s servers accepted it.

Client-Committee

The committee responsible for starting up/closing up the voting system by sending Grpc messages to one server from each state. The committee uses group membership through zookeeper to find this servers. The server that got the start/close message will send it to all of the members of his state. There is no need for total order because there is only one client that can send start/close messages. In case not all of the states accepted the start/close message the start/close action failed and the committee will have to try again.

In addition, the committee responsible for gathering the results, again by sending Grpc message to one server from each state, each one of tham has all the votes of the corresponding state because all of the servers of state has the same view of the votes.

Zookeeper

We use zookeeper to implement membership and leader election.

We are maintaining the zookeeper’s file-system in the following way:

* When a new node/server joins state, he joins under membership app to his state and his znode name will be compose from the ip and port of the node.

The same process occurs with leader election app except for one difference: the znode joines to the membership app as ephemeral sequential node while he joines to the leader election app as ephemeral node.

* When the zookeeper didn’t get heart-beat from a znode for a pre-defined time, then he start a process of choosing a leeder for the node’s state. If the leader is still alive he will get to stay the leader. But if the leader fall the new leader will be the node with the lowest sequential number.
* Each state’s znode will have a watch that keep track of the changes in the state.
* Every node knows which nodes are in his state and who is the leader.

**Zookeeper filesystem hierarchy and applications implementations:**

