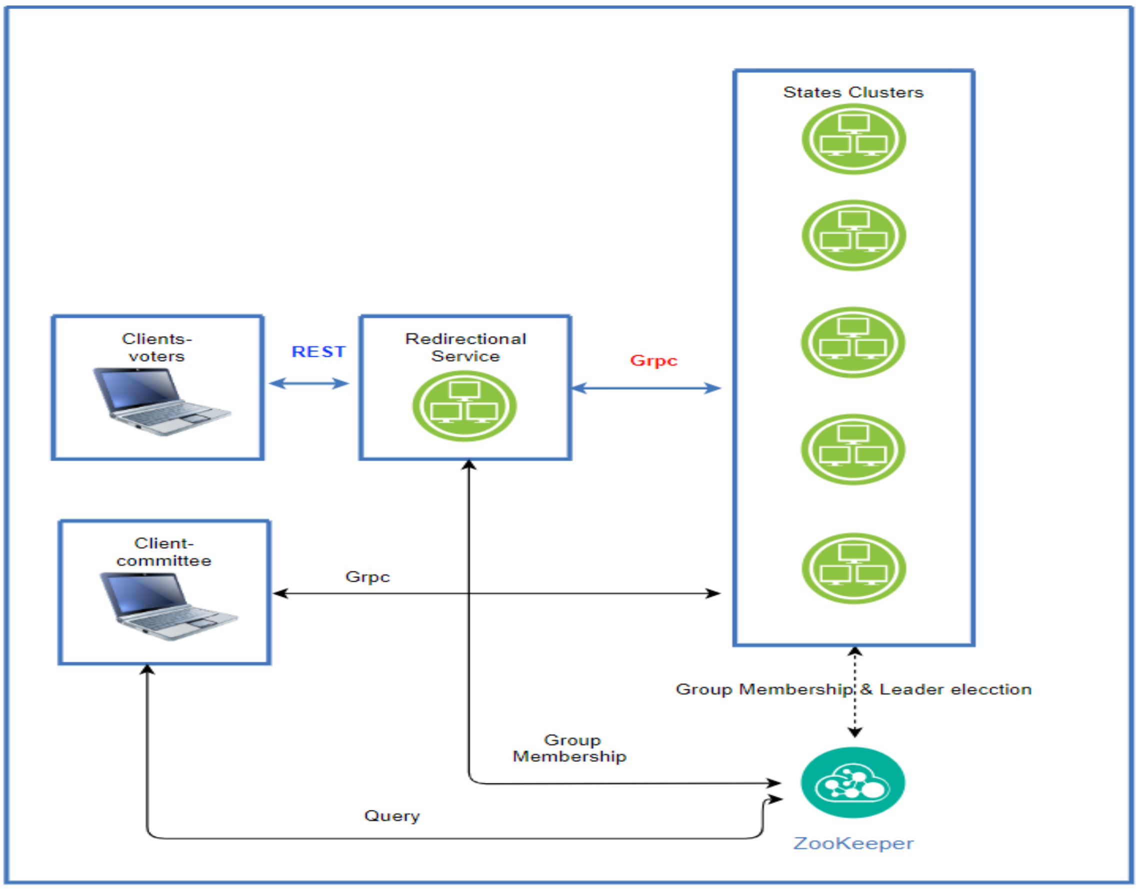
Election in the USA – Distributed Systems Project – Documentation:

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.
- **High consistency –** We create a replica of each user vote on each of the state nodes, and we don't return a reply to the client until all the state cluster have the most recent vote.

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. It receives a vote from the clients via REST and sends 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 at random 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 their 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 will help us add a firewall like functionality where we can decide if we want to move the user request to the state clusters which have the election DB on them, thus reducing the state clusters load.

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.

We will also note that the state cluster is decentralized which means every node has the exact same functionality, so there is no single point of failure.

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 leader election.
- Also, in this design we promise atomicity since we won't ack a client request until all 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.

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 (HTTP 500 – internal server error).
- 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 check the timestamp on the vote and if it's a newer vote he will update the vote and try to reply by sending back to the leader if he succeeded.
- This implementation creates 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 the state's cluster sees the same view (linearizability).
  In addition, this implementation establishes atomicity by accepting vote request only if all the cluster succeed to update it. Meaning, there is only two options: either all the cluster's servers accepted the vote or none of the cluster's servers accepted it.

Client-Committee

The committee responsible for starting or stopping the voting system by sending Grpc messages to one server from each state. The committee establish a short time connection to the Zookeeper only to query it regarding servers in the state clusters. The server that got the start/close message will send it to all the members of his state. There is no need for total order because there is only one client that can send start/stop messages. In case not all of the states accepted the start/close message the start/stop action will fail, 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 them 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 using the Zookeeper in the following way:

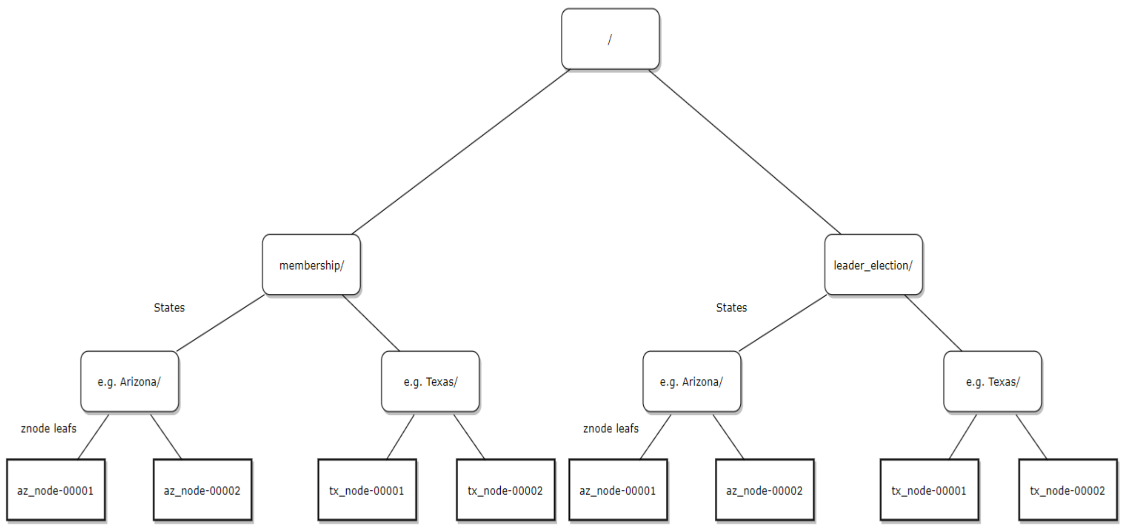
- When a new node joins state, he joins under membership_app to his state as a ephemeral Znode and its name will be composed from the IP and port of the node.
  Similarly, the node created an ephermeral sequential Znode under leaderelection_app under his state persistent Znode.

- After the nodes is created, we create a watch for changes under the state persistent node under both membership app and leaderelection_app.
  When there is a change under the state Znode we update the local membership data each node has and update the leader (If it is the one that failed).

Technical justification:

- We chose this way to implement leader election because we want that each node will know who the leader of the cluster is to provide the requirements for the voting algorithm.
- We lower the amount of traffic that goes through the Zookeeper by having each node store the data it needs locally and only change it in-case a watcher event as occurred.
- We also decrease the amount of requests from the zookeeper by creating the Znodes of the a node under the membership_app with it's name being his IP and port, so we don't have to request the data of the Znode as well.

Zookeeper filesystem hierarchy and applications implementations:



A short analysis of our system fault tolerance and availability:

Let's assume each cluster has N nodes, to give an understanding of how fault tolerant each cluster is:

State cluster - A state cluster only stop working when all the nodes under it fails meaning, for N nodes we can handle $N-1$ failures.

Zookeeper – Just as a reminder the Zookeeper requires a majority to function correctly so for N Zookeeper nodes we can handle $Cieling\left(\frac{N}{2}\right)-1$ failures.

Redirection cluster – We treat this as a cluster, but they are all independent from one another and you can work with only 1 of these, so the fault tolerance of this cluster is $N-1$ as well.

Availability - Our design doesn't focus on availability, but rather on consistency. By that we mean that if a server in a cluster have crashed in some part of the voting process the cluster will take a few seconds (depending on the zookeeper timeout we choose in advance) to recover, and in the time the cluster recovers no votes in that state will be recorded, but note that since we created another layer with the redirection service. It is only the state cluster that had a failed node that will suffer from that downtime.

In conclusion: *While this design will most likely provide high uptime since servers shouldn't be prone to crash a lot. We don't provide the highest possible availability, but we do offer very high fault tolerance, and very high consistency, which from out engineering point of view is much more important for an election system.*

Appendix 1 – Rollback failure

As we discussed above if one of the servers in the cluster is noted as available in the Zookeeper membership, but the leader isn't able to connect to it, we decided that we will ask the user to try and vote at a later time, and rollback to the previous state before his or her vote.

This potentially create the following issue: Let's say we have 3 active members in the cluster, were able to update 2 of them already and for some reason the connectivity with the 3rd one is broken for a brief moment, but it is still alive in the zookeeper eyes. We will then by the way of the algorithm we designed will ask the two servers to rollback the vote they recorded.

And here we could potentially with a very low probability have an issue of data consistency if one of the 2 server didn't get the rollback message after the timeout we set expired, but it is still alive. This is a **design flaw** we discovered pretty late into development and we have thought about several ways to fix it, but the good solutions (*) will take a lot of times to implement and since we are short on time we came up with a solution which is easy to implement but decrease the fault tolerance of the system we created.

The solution: Each node in each cluster will have a parameter we call data integrity. If the race condition above happened we will create a persistent node in the zookeeper under data integrity app which will contain nodes that aren't to be trusted without manual check (we log all the activity so the administrator can later check if the data_integrity is really flawed). Then when checking for results from a specific cluster we will remove nodes which have their integrity in question. Since the probability of this event is low, we concluded that this shouldn't happen much, so while we decrease our system fault tolerance with this solution, we still maintain its correctness and with low cost on development time.

Our other option was to kill the server which we suspect, but with this method we provide a more redundant approach, since even if we suspect the server it could still have valid information, but this needs to be checked only in case we have no other server we can trust.

(*) For e.g having a background process thread save all the votes that caused it and try periodically to rollback these events.