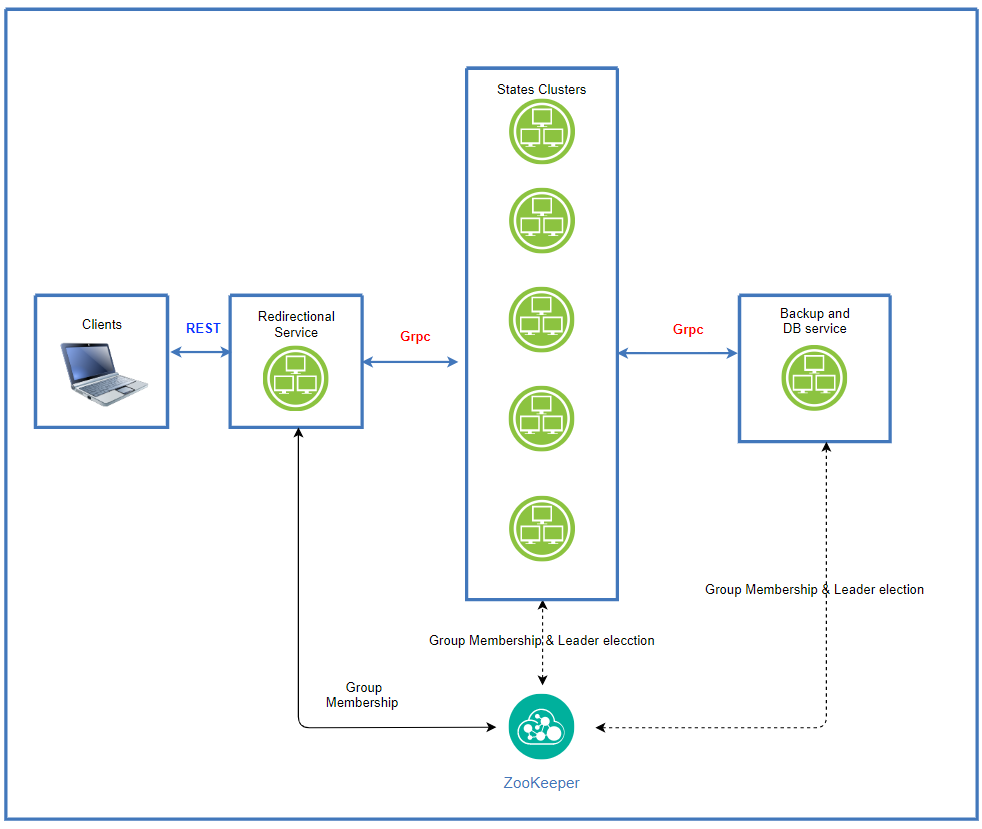
**Election in the USA – Distributed Systems Project – Documentation:**

**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 to the correct state cluster depending on the voter information. It will use group membership of the state 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 incase 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 (

**Backup and DB service**

We use this layer to gather all the information from the states on the current voting state, this cluster will also provide fault tolerance mechanism, since it will work just as the state clusters by using a master-slave methodology.

Its main purpose is to provide the clients with the election results while keeping this service separated from the states clusters.

It will work in the following way – after a client sends a request for election results the redirection service will move the request to the Backup and DB service which in turn will query a member from each cluster of a state for the current election in the state, and it will return to the client the information after it gathered it all and replicated on a quorum of servers.

**Technical justification:**

* This service is mostly made to satisfy the requirement that the clients can see the election results at all time.
* In addition implementing this as a different service increase our fault tolerance capability since if for some reason one of our state clusters exceeded the limits of failures it can withhold we will be able to partially reconstruct the cluster state using this service, this isn’t implemented in our project, but can be easily implemented because of the separation.

**Zookeeper filesystem hierarchy and applications implementations:**

