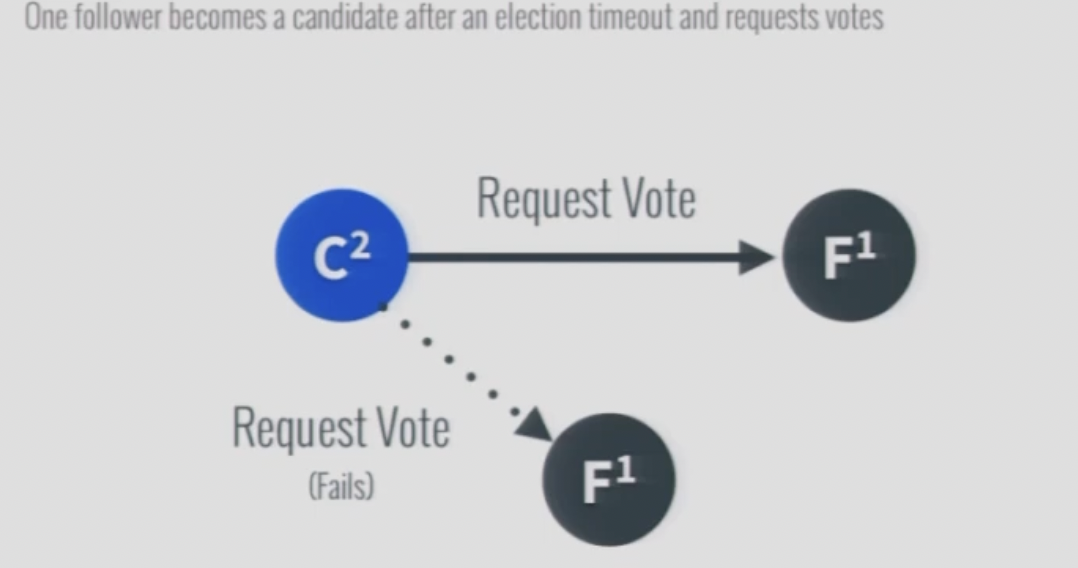
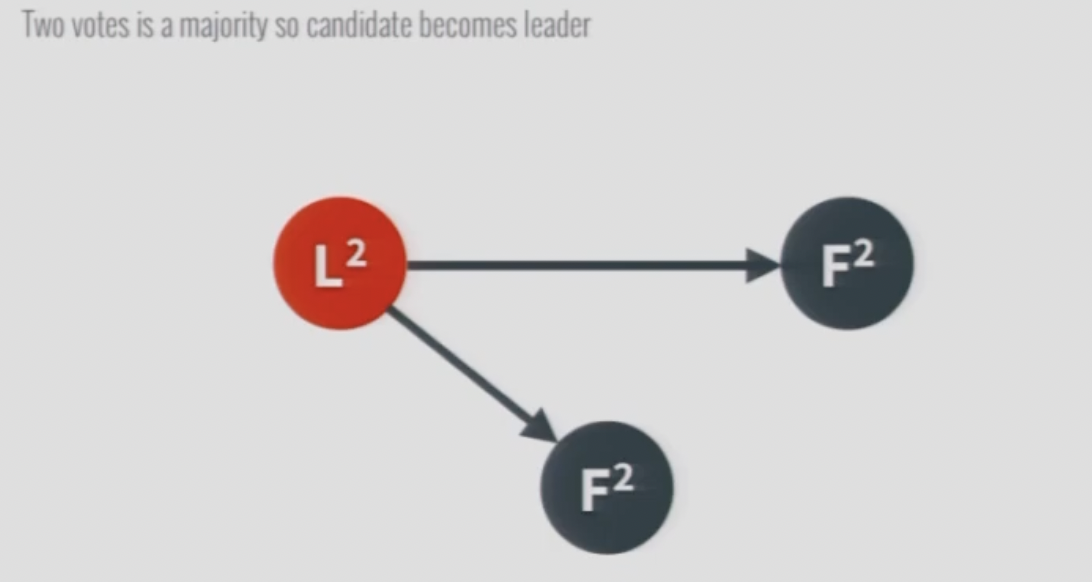
**Distributed Consensus and Data Replication**

One Algorithm that solves this is Raft

* **Leader, Follower and Candidate**. If the Followers don’t listen from leaders for a while (heartbeat), one of them times out and becomes a candidate. Timeouts could be random to make sure not many transition to candidate at the same time. Candidate asks for votes to become a leader, majority should agree. Once it becomes a leader, sends log entries which is the state of the system replicated to the followers. Also, sends heartbeats.
* **Leader Election: election term. Good case:**



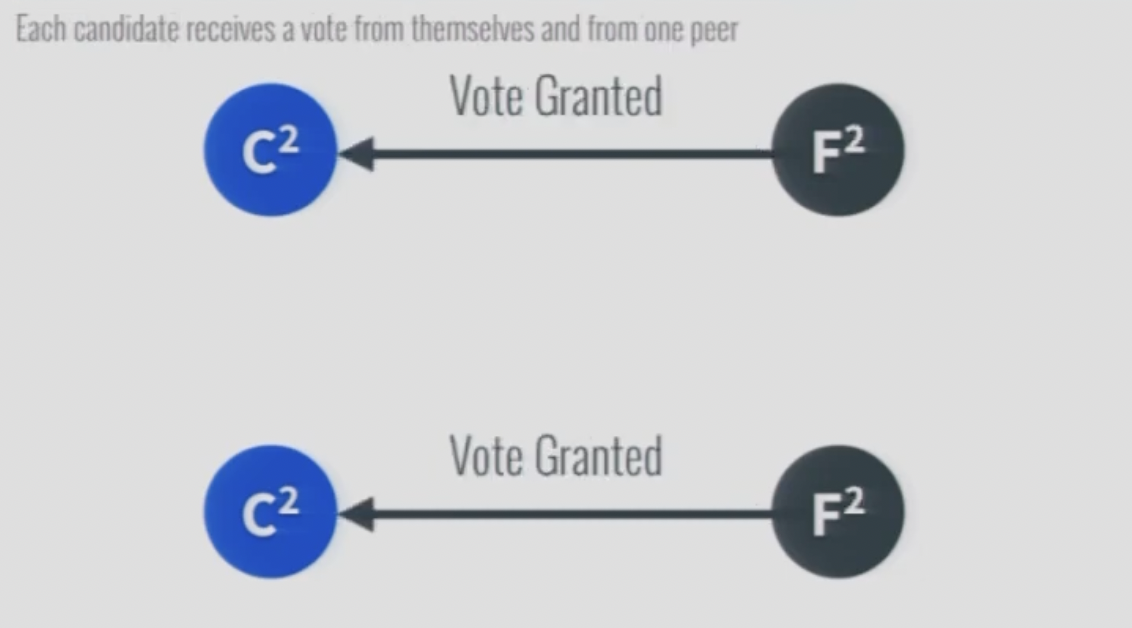
The leader sends the heartbeat saying that it’s a leader with the updated term, so the second follower accepts it and updates its term.



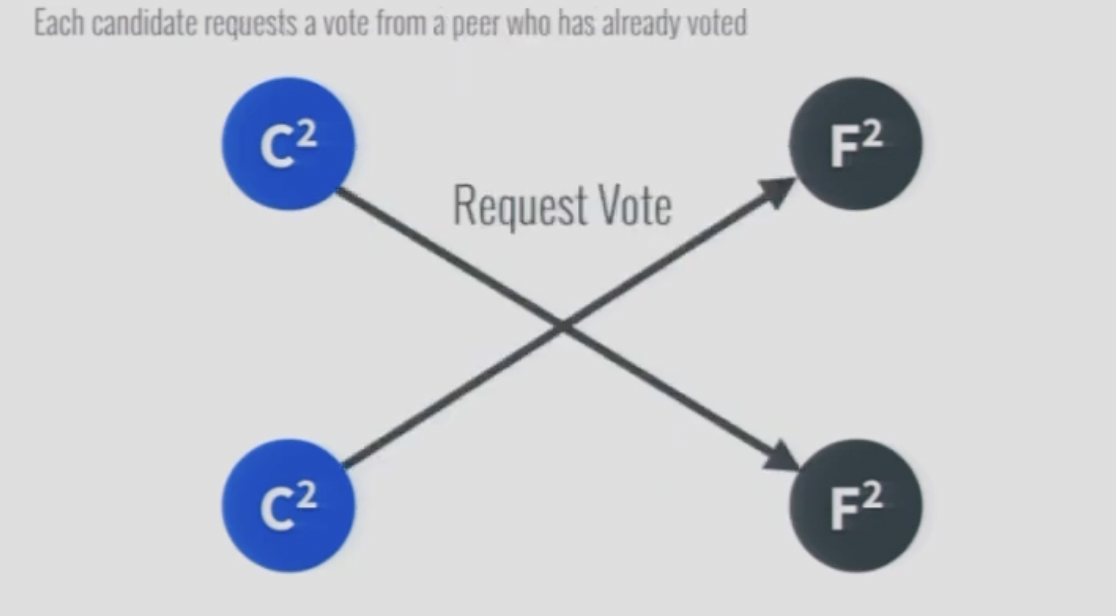
* **Leader Election: Split Votes:**

Diagram

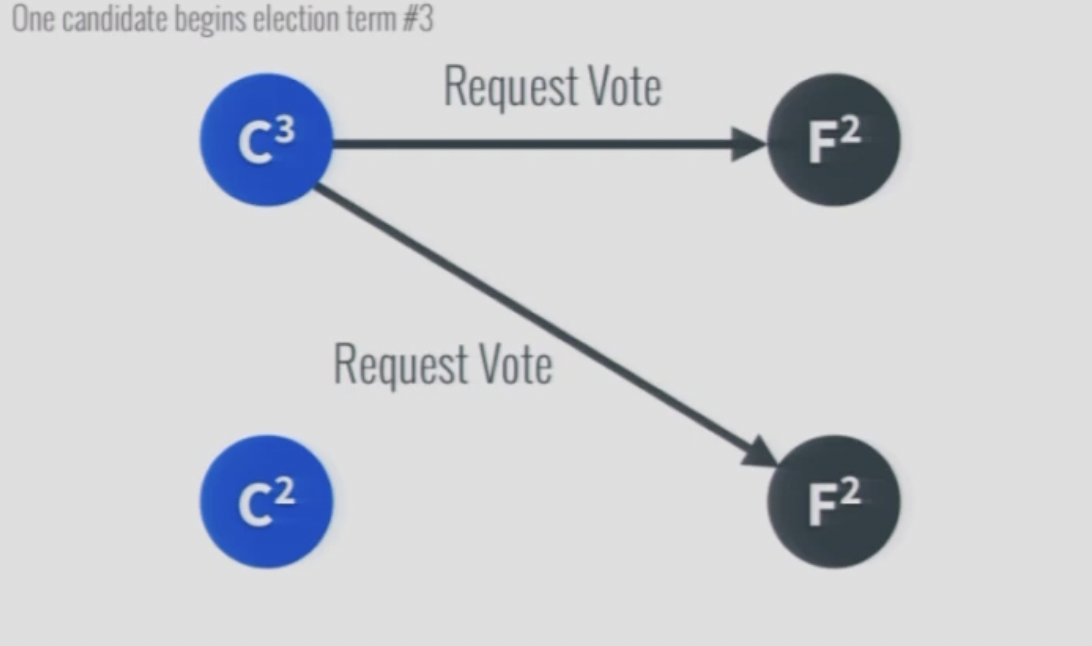
Description automatically generated



Votes in the same term so the followers deny the cross vote.



Now one of the candidates, will randomly timeout, start a new term and this will transition to the good case

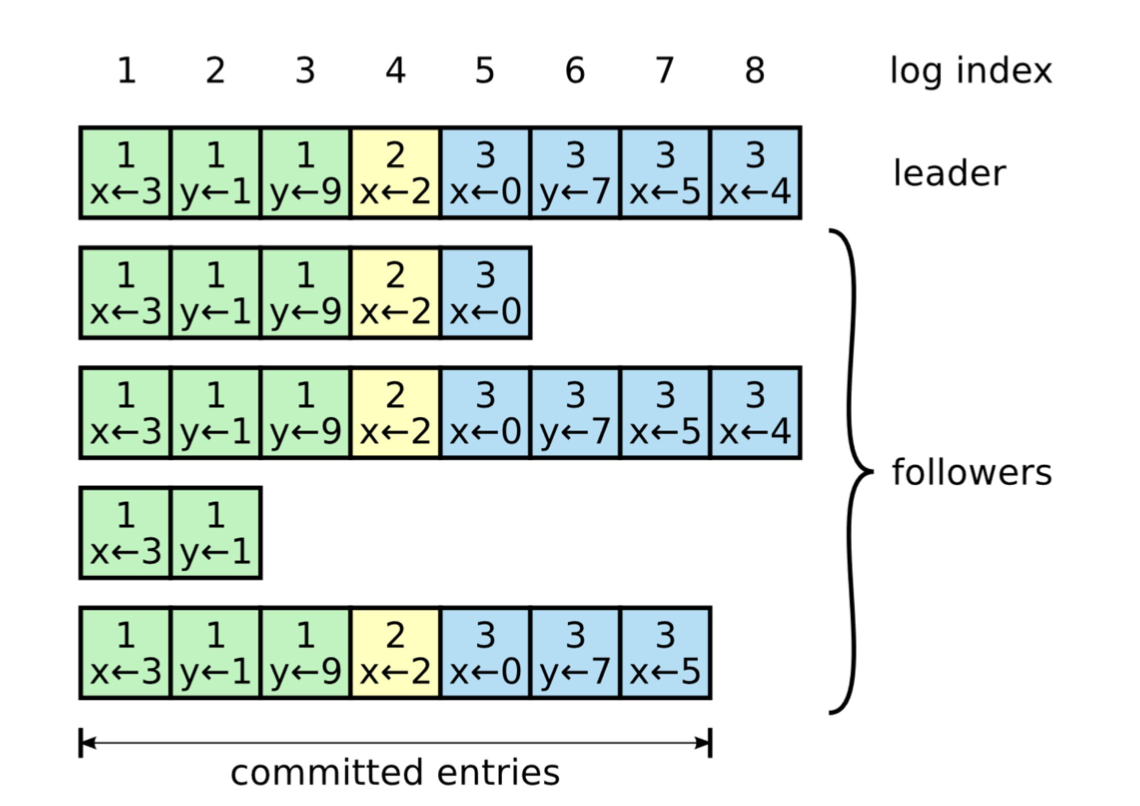


The entire process can also be visualized using the following state machine diagram:

Diagram

Description automatically generated

* **Log or Data Replication:**
  + The leader sends Append Entries to the followers, every time some update happens.



* The followers responds with OK and adds the entries in their logs. Only when the Leader receives acknowledgements from majority, it considers the entry to be committed and applies the update to the system i.e. local data store
* The leader then sends AppendEntries requests to the followers with the highest committed index and A follower only applies a log entry to its local state when it finds out that the leader has committed the entry.
* Network Partitions: If the leader dies, only the followers with the most upto date logs can become a leader. This is because a process won’t vote for a candidate with less upto date log than itself. This guarantees that the elected process has all the committed entries. If the follower fails and comes back up, it’s possible that an appendEntries request is rejected by the follower because it creates a hole in it’s log. In that case leader retries with more log entries till the AppendEntries request is finally accepted by the follower.
* **Consistency Models:**
  + **Strong Consistency:** Only leader accepts reads and writes. What if the client sends a read request to the leader and by the time the request gets there, the server assumes it’s the leader, but it actually was just deposed? If the ex-leader was to process the request, the system would no longer be strongly consistent. To guard against this case, the presumed leader first needs to contact a majority of the replicas to confirm whether it still is the leader. Only then it’s allowed to execute the request and send back the re- sponse to the client. This considerably increases the time required to serve a read.
  + **Sequential Consistency:** String consistency has limitation of a single choke point – less throughput.In sequential consistency, the observers connect to respective replicas, so even though their observed states are different, they evolve in the same order for them.
  + **Eventual Consistency:** The only guarantee the client has is that eventually, all followers will converge to the final state if the writes to the system stop
* **CAP:** If partition has happened, The system has two choices when this happens, it can either:
  + remain available by allowing followers to serve reads, sacrificing strong consistency;
  + or guarantee strong consistency by failing reads that can’t reach the leader.
* **PACLEC:** If partition has not happened, choose between strong consistency and low latency. If it’s strongly consistent, then you need to query quorum of nodes for example, before returning a read request. This increases latency.