

Raft and Paxos: A brief Introduction to the Basic Consensus Protocols Powering the Distributed Systems Today

Diwakar Bhardwaj

9-12 minutes

Can we all agree..... ?

Many of the institutions we depend on as a modern society like hospitals, banks, stock exchanges and airports, need highly reliable and available computer systems to work. Unfortunately, single machines are not up to the task; they're susceptible to crashes and downtime, and need regular maintenance. The solution, of course is to have multiple machines doing the same job, but how do we get them to do so? This is not a simple problem; the network connections between the machines are not reliable, the machines themselves may crash and we don't want to involve people to fix things at 3 AM on a Sunday. Consensus is a fundamental part of building replicated systems and getting these machines to work together. The idea of getting them to work together is to have them all store the same data or commands in their logs. This increases the overall system reliability, because if one machine fails, another machine can take up its work.

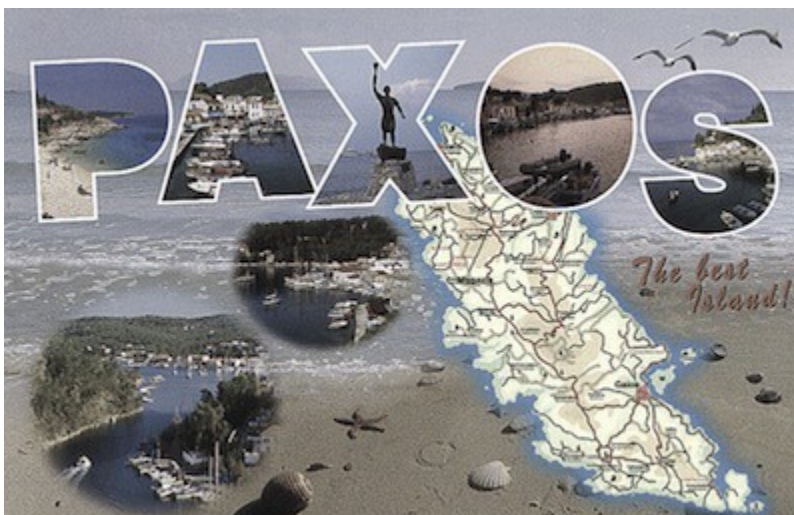
Consensus helps us in solving a number of problems in distributed systems. Some of these are:


1. **Leader election:** A group of servers electing a leader amongst themselves and letting everyone know about it. An example of this would be a leader election in etcd. etcd is a distributed key value database used by Kubernetes. Both etcd and Kubernetes are backbones of microservices infrastructure in Sixt.
2. **Mutual Exclusion:** A group of servers ensuring mutually exclusive access to a critical resource.
3. **Membership failure & detection:** A group of servers maintaining a list of each other, updating it as someone enters or leaves the group.
4. **Reliable multicasting:** A group of servers attempting to receive the same updates in the same order as each other.
5. **Load balancing:** Distributing incoming network traffic across a group of backend servers.

Other problems include maintaining a distributed ledger in blockchain technologies in cryptocurrencies, clock synchronisations between servers etc.

There are a variety of consensus protocols available which solve these problems directly or indirectly. We'll have a look at two of them going ahead.

Paxos





Paxos is the oldest consensus protocol published way back in 1989 by [Leslie Lamport](#). Paxos is used in multiple graph databases like neo4j. It is also used in column store database Cassandra for leader election.

Let's take a real life example of how Paxos works:

There are 4 friends named Rajesh, Rohit, Suresh and Vijay hanging out at Rohit's place. Rajesh is listening to music with his earplugs on, and not listening to what others are saying. Suresh suggests that they all should go for a movie, and Vijay agrees with him. Then, Rajesh removes his earplugs and suggests they all should go for bowling. Rohit agrees with him, and Vijay changes his decision and agrees with Rohit for bowling. Since the other three agree on bowling, Suresh has no choice but to go for bowling.

Paxos algorithm

There are 3 roles in Paxos:

1. Proposers (Suresh and Rajesh)
2. Acceptors (Vijay and Rohit)
3. Listeners (Rajesh after a consensus reached on bowling)

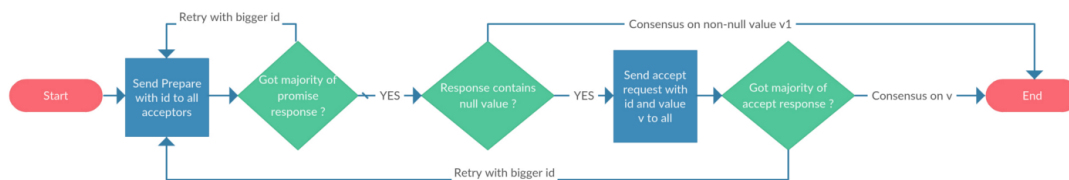
As you can see above, any actor can take on any role at a given point in time.

Proposer wants to propose a value ***v*** and wants the system to reach a consensus on this value. Hence, it sends a ***prepare*** request along with an ***id*** to all the acceptors and expects a ***promise*** response along with a value. There can be 3 cases here:

1. If he gets a majority number (quorum) of promise response with null value ***v*** (This means there is no other value ***v1*** onto which a consensus has been reached and he can propose his own value ***v***), he sends an ***accept*** request which contains a this value ***v*** and

previously accepted ***id*** to all the acceptors. If again he gets a majority of responses with same ***id*** and value ***v***, it indicates that a consensus has been reached on value ***v***. This is what Suresh did above

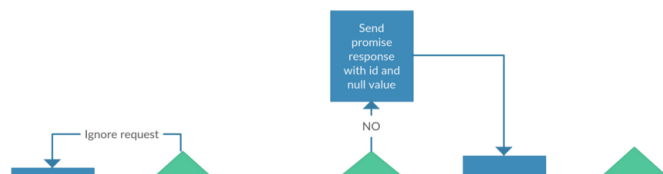
2. If he gets a majority number (quorum) of promise response with a different value ***v1***, that means a consensus has already been reached on a value proposed by a different proposer and it will accept value ***v1*** as well. This is similar to what Rajesh did in our example above.
3. If he doesn't get a majority number of promise response, he will try with a higher ***id*** again.

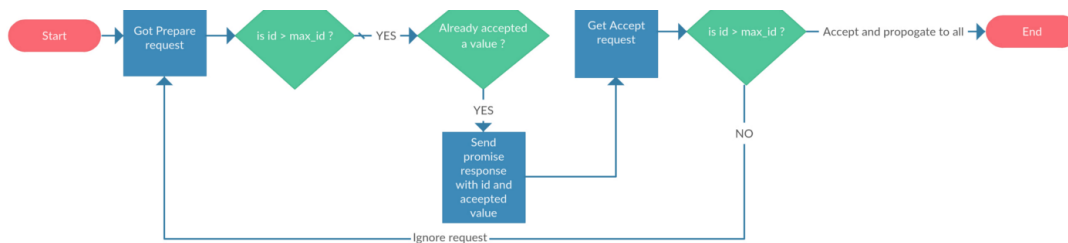


Typical Proposer flow

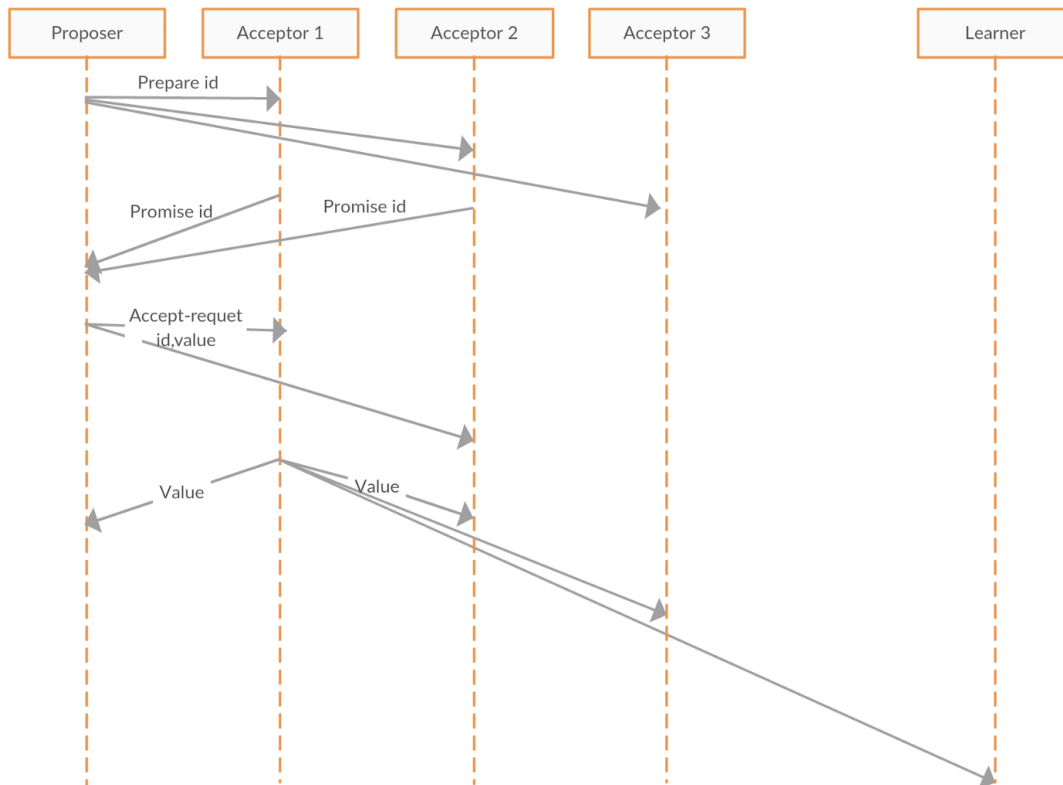
Acceptors maintain a couple of fields in their local memory, a maximum id (*max_id*) and already accepted value(*v*).

1. If the ***id*** proposed by the Proposer is higher than the ***max_id*** and the Acceptor has not accepted any value so far, then it accepts that ***id***, returns the response with null value and stores this ***id*** as the new ***max_id*** in its local memory.
2. If the ***id*** proposed by the Proposer is higher than the ***max_id*** but the Acceptor has already accepted a different value ***v***, then it accepts that ***id***, returns the response with value ***v*** and stores this ***id*** as the new ***max_id*** in its local memory.
3. If the ***id*** proposed by the Proposer is lower than the ***max_id***, then the Acceptor will ignore this request.



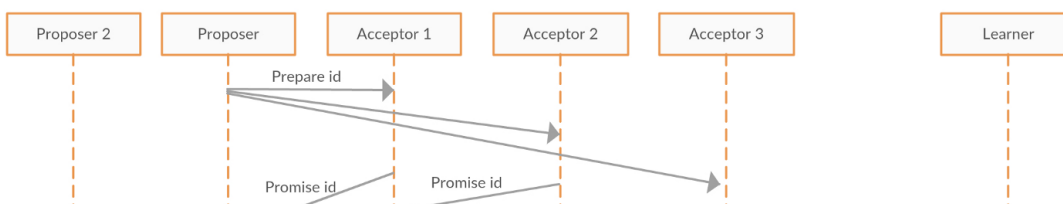


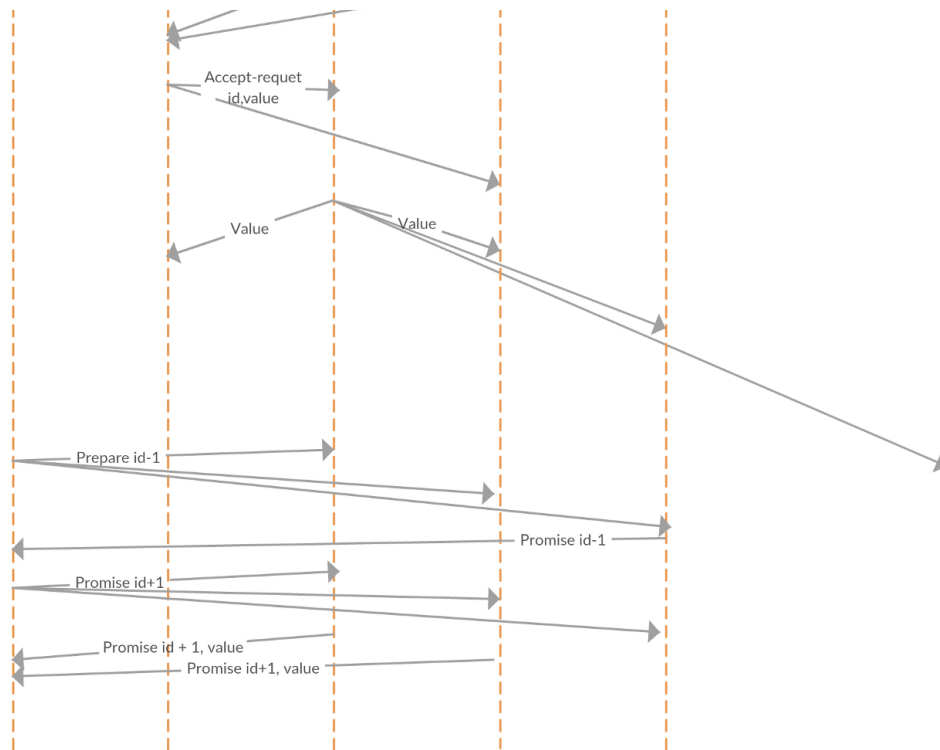
Let's consider another example to understand a basic interaction between all these roles



As you can see, the Proposer proposes id, it gets accepted by a majority, and then the Proposer proposes a value with that id. Please note that Acceptor 3 has not responded with Promise id. A consensus has been reached for this value and this value is propagated to all the nodes including listener.

Now, let's add another Proposer to this whole setup. Also, now Acceptor 3 has woken up





New Proposer 2 sent a Prepare request with an id which is lesser than the accepted id. Acceptor 1 and 2 ignored this request. Acceptor 3 which has just woken up accepted it and sent a Promise. But since it was the only one which sent a prepare and a majority has not sent a Prepare, Proposer 2 ignored that response.

Now Proposer 2 sent another Prepare request with an id which is higher than the accepted id. This time, Acceptor 1 and 2 responded but with an accepted value. After receiving the promise response with a value, Proposer 2 identified that there is an already accepted consensus on value and it accepted that value as well.

There can be a case of infinite loop here, when 2 proposers keep outbidding each other with higher id. They will not give enough time to each other to send accept-request to acceptors. We can solve this problem by waiting for a backoff period before sending the next Promise request.

Raft

Raft can be described as a simpler version of Paxos. It was

designed for being more understandable than Paxos. It is a fairly new protocol, being developed in 2014. Raft is used in etcd, consul, docker to name a few. All of these systems are used in Sixt.

Raft decomposes consensus into leader election and log propagation phases. After leader election, leader takes all the decisions and communicates to other nodes through ordered logs.

There are 3 possible states for a node in Raft consensus protocol:

1. Leader
2. Follower
3. Candidate

All the nodes start off with a Follower state. If followers don't hear from a leader then they can become a candidate. The candidate then requests votes from other nodes. The candidate becomes the leader if it gets votes from a majority of nodes. This process is called a **Leader election**.

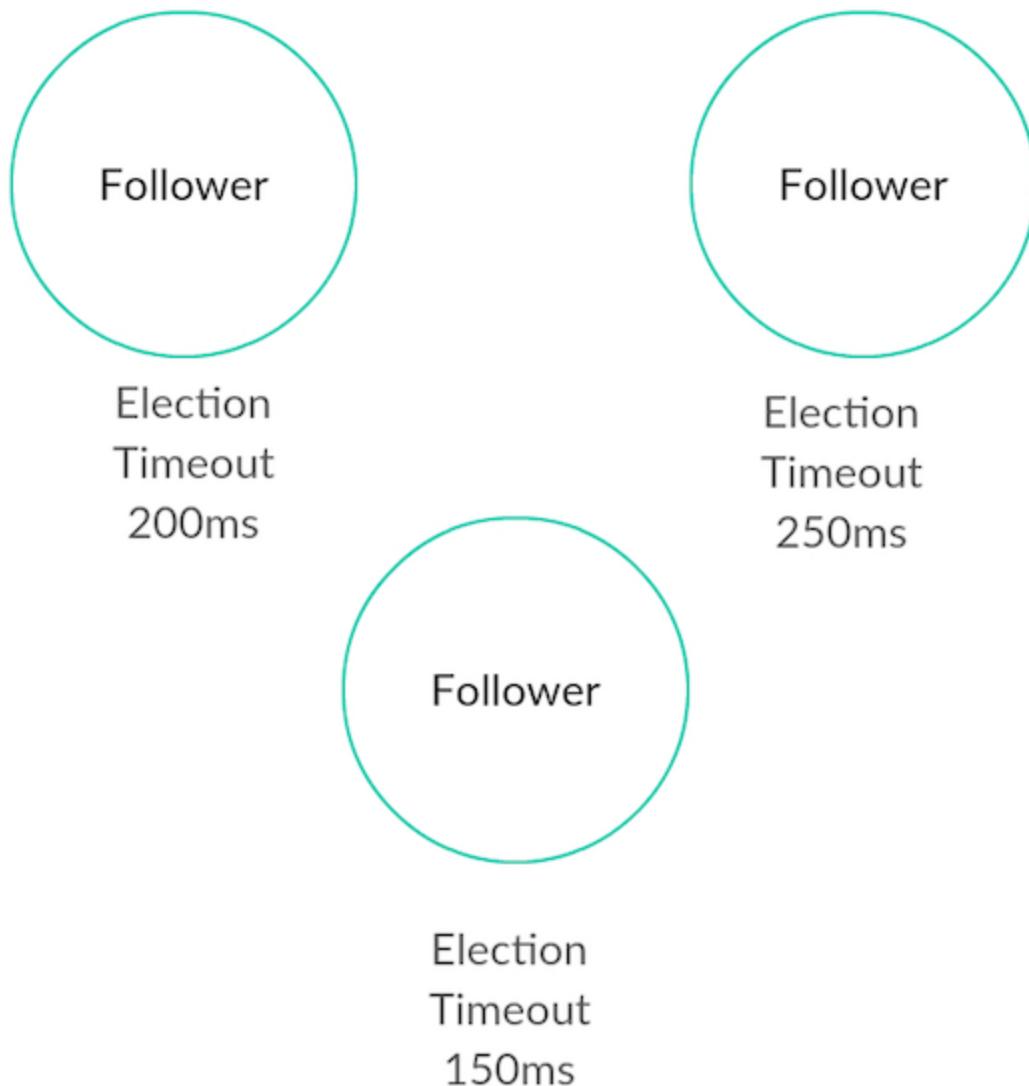
All changes to the system now go through the leader. Each change is added as an entry in the node's log. Values remain in uncommitted state and get written to the log first, and then these logs are replicated to other nodes. Once a majority of nodes respond, the value gets committed to the leader node as well. The leader then notifies the followers that the entry is committed, so that the followers can commit the value as well. The cluster has now come to consensus about the system state. This process is called **Log replication**.

Leader election

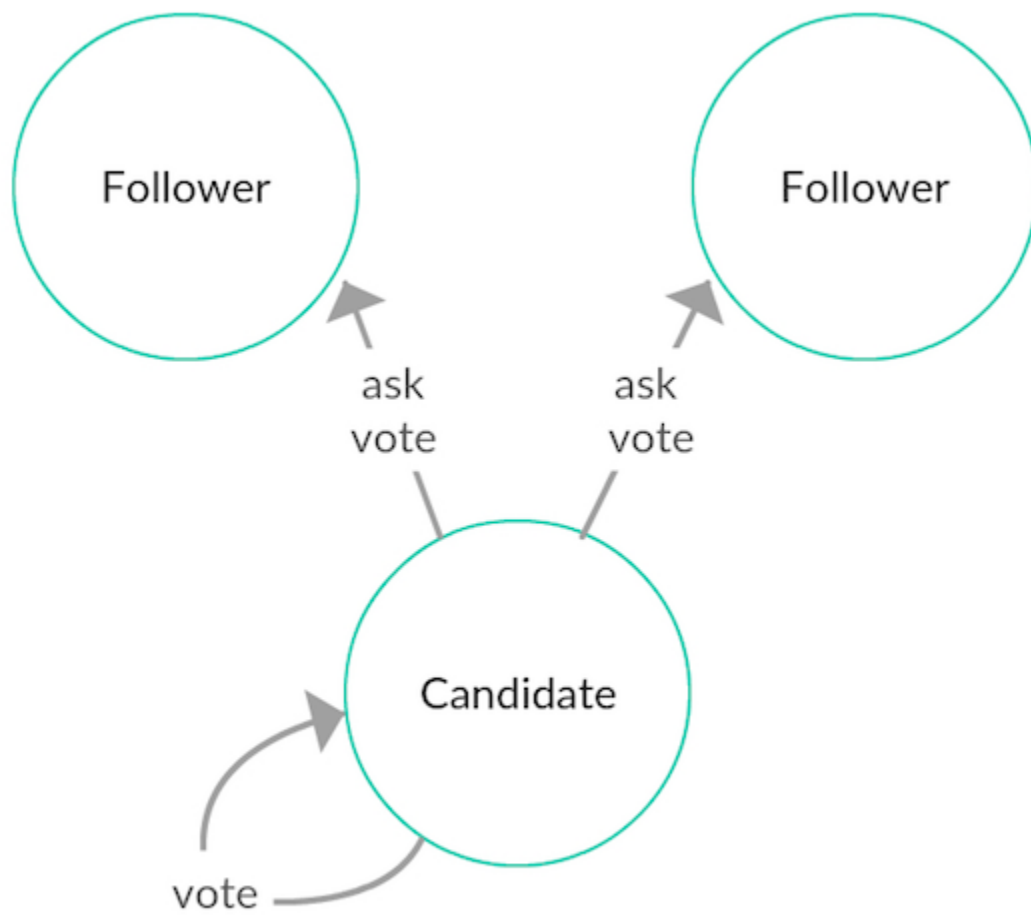
Every follower waits for a certain amount of time before becoming a candidate. This time is called the **election timeout**. After the election timeout the follower becomes a candidate and starts a new **election term**. It Votes for itself and sends out Request Vote

messages to other nodes. If the receiving node hasn't voted yet in this term then it votes for the candidate and the node resets its election timeout. Once a candidate has a majority of votes it becomes leader. Leader keeps sending regular messages to followers to keep its status as leader. This message is called a **heartbeat message**. If a follower doesn't receive a heartbeat message in a certain time interval, it can become a candidate and the whole cycle of leader election will repeat.

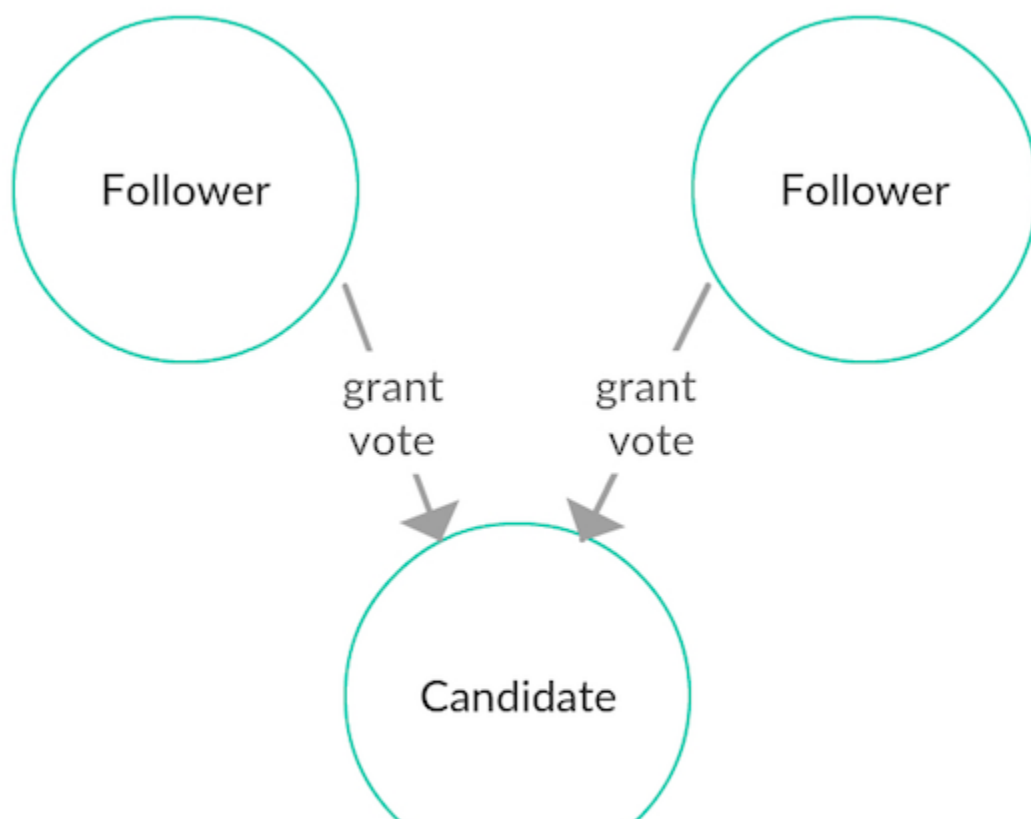
For example, there are 3 nodes acting as a follower. Each has a different election time out window.




Follower with the lowest election timeout value becomes a candidate and asks for vote.

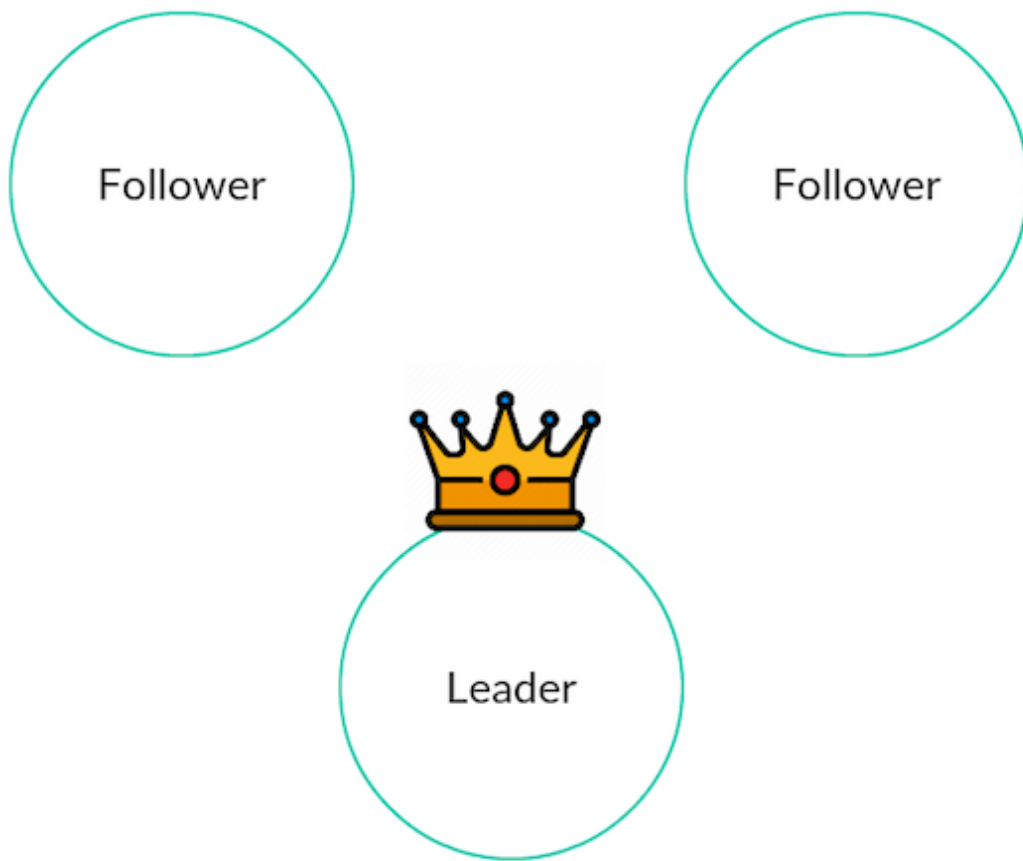


After the votes has been granted, candidate becomes leader.





Now, when everybody has voted. We have a leader.



Log replication

Once we have a leader elected, we need to replicate all changes to our system to all nodes. This is done by using **Append Entries** messages to the followers.

It works like this:

1. Client sends a change to the elected leader.
2. The change is appended to the leader's log.
3. Then the change is sent to the followers on the next heartbeat.
4. An entry is committed once a majority of followers acknowledge it and a response is sent to the client.

This is usually called a **2 Phase commit**.

There are other protocols that solves different problems.

1. ZAB (Zookeeper Atomic Broadcast) was used in zookeeper for setting a master-slave system.
2. Gossip protocol is used in consul for membership detection and failures.
3. Viewstamped Replication is another state machine replication protocol, and it provides consensus too.

References

1. <https://raft.github.io/raft.pdf>
2. <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>
3. <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>
4. https://www.youtube.com/watch?v=_KUV-34fg3A