Leader Election Algorithms Implementation

Martina Lupini
dept. Computer Engineering, University of Rome Tor Vergata
Rome, Italy
martina.lupini@alumni.uniroma2.eu

***Abstract*—In a distributed system it is often required to have one process to act as the leader. The election of the leader can happen in a distributed way among the processes by using leader election algorithms. The objective of this work is the implementation of two leader election algorithms, in particular, the Chang-Roberts algorithm [1] and the Bully algorithm [2].**

I. Introduction

Leader election is a process in distributed systems where a group of nodes collaborates to choose a leader among them. Leader election algorithms have the following requirements:

1) All processes have the same initial state and there is no designated leader a priori.
2) Every process executes the same local algorithm.
3) The properties of liveness and safety need to be respected.
4) The final result does not depend on the process that initiates the election.
5) Each process has a unique ID.

We are going to examine the algorithms in detail. In the report the terms *process* and *node* will be used as synonyms.

A. Chang-Robert algorithm

The Chang-Robert algorithm requires that the processes are organized in a logical ring and each process has to know at least its next node in such ring. The algorithm can be executed in asynchronous systems and is an improvement of LeLann's algorithm. Assuming that the messages are passed unidirectionally, the algorithm solves the election problem with an average of $O(Nlog(N))$ messages (where N is the number of processes involved).
When a node detects no leader or that the current leader is not working, it initiates the election by sending a message containing its ID to the next process in the ring. If the next process is not working it will contact the process after the next process and so on until there is a working node.
When a process receives a message, it compares the ID contained in the message with its ID. If the received ID is bigger, it forwards the message to the next node. If the ID is smaller it drops the message and starts a new election. If the ID is the same that means that the process has the biggest ID in the system and becomes the leader. When a node wins an election it informs all the other processes that it is the leader.

B. Bully algorithm

This algorithm does not require a specific topology but the distributed system needs to be synchronous in order to detect faulty processes by using a timeout mechanism. Also, each node must have complete knowledge of the other processes in the system. The complexity of this algorithm is $O(N)$ in the best case (the node that starts the election is the one with the highest ID) and $O(N^2)$ in the worst case (the node that starts the election has the smallest ID).
A round of election starts when a node detects no leader or that the leader is not functioning. The node then sends an election message to all the processes with higher IDs. If at least one of them responds the process loses the election.
When a process receives an election message from another node with a lower ID, it responds to it and then starts a new election. In the end, the process currently functioning with the highest ID is elected. After winning an election, the process sends a message to all the other nodes informing them it is the leader.

II. Background

The programming language used in the implementation is Go. To ease the deployment and increase portability, the main application components are containerized using Docker. In particular, each process (i.e. the nodes and the registry) corresponds to a container. The containers must be deployed on a single host and coordinated with Docker Compose.
The communication between the processes is made possible by using GoRPC and the bridge driver offered by Docker.
The containers can be deployed on a local computer or on an AWS EC2 instance.

III. Solution Design

The architecture is composed of three main components: the registry, the backup registry, and the nodes. This implementation assumes that the backup registry has an extremely low, almost non-existent, probability of crashing.
The registry and the backup have a known hostname and port number and distinguish themselves using IDs: 1 for the main registry and 2 for the backup. The architecture takes inspiration from Kafka (and in particular from the design of the brokers). The main registry represents the entry point of the system and it's the one that the nodes contact by default. The other registry acts as the backup in case the main registry is not working and it is perfectly in sync with the latter.
The algorithm can be selected using a configuration file and if no algorithm is selected the default is Chang-Roberts algorithm.

The nodes' code contains the implementation of the algorithms. Each node does not have a hardcoded hostname and port but it will retrieve them at runtime.
The state of the registry comprehends:

- The information relating to the peers in the system (in particular hostname and port number of each one of them).
- The IDs already assigned.
- The chosen algorithm.
- The ID of the registry.

The state of each process comprehends:

- ID.
- Hostname and port number of the node.
- The addresses of the peers in the system.
- The chosen algorithm.
- The leader.

A. Starting the registries

The registries are started before the other processes. In particular, the backup registry is started before the main registry. After starting, the backup registry listens for incoming requests.
When the main registry starts, before listening for requests, it contacts the backup registry with the RPC `RetrieveInfo()` to obtain the list of peers and assigned IDs in the system. Of course, the first time the registry is started these lists are empty, but this step becomes essential in the following scenario. Let's suppose that the registry fails after a number N of nodes have joined the network. At this point, the backup registry takes the lead and registers the new T nodes that want to join the network. If the main registry becomes available again it can get the changes in the system by contacting the backup registry.
The backup registry has also a routine in the background used to check if the main registry is alive. The routine is started after receiving the first message from the main registry.

B. Entering the network

When a node enters the network it contacts the main registry with the RPC `AddNode()` to obtain the addresses of the nodes already in the system, the algorithm to use and an ID. If the main registry is not available then the backup registry is contacted. As said before, in the Bully algorithm the processes have complete knowledge of all the other processes in the system. In the Chang-Robert algorithm, this is not required but the implementation demands it to be more fault-tolerant.
During the addition of the node in the system, the main registry contacts the backup registry to keep it in sync using the RPC `Sync()`. After the registry has given all the information needed, the new node contacts all the other processes by calling the remote procedure `NewPeer()` and informs them of its presence in the network.

Each node has a routine in the background called `CheckLeaderAlive()` to check if the leader is working.

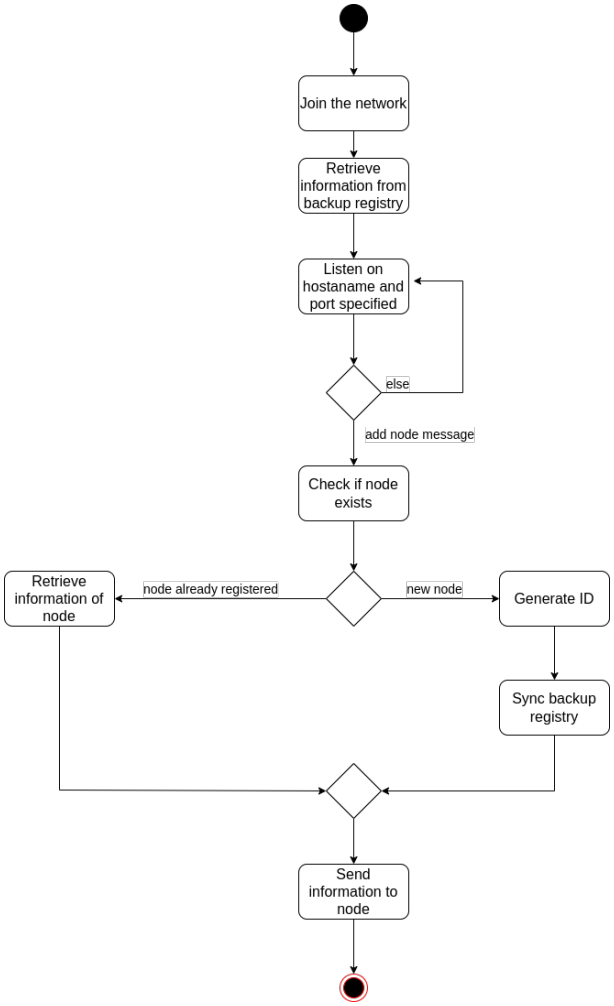

Fig. 1. Actvity diagram of the main registry operations

This is done by trying to connect to the leader within a timeout. If the attempt fails, then the leader is considered to be unavailable and the node starts a new election.
Of course, after joining the network the node has no leader so it starts an election through the function `ElectionCR()` or `ElectionBully()`, depending on the algorithm chosen.

C. Election in Chang-Roberts' algorithm

The function responsible for the election is `ElectionCR()`. The main goal is to identify the next available node in the logical ring and send the election message. It also sets the flag `election` to inform the routine in the background that an election is in progress and there is no point in attempting to ping the leader.
The node sends an election message to the next node containing:

- ID of the node that started the election.
- The number of elections started by the node (to distinguish between one election and the other).
- The phase of the election, which takes track of the nodes that the message has passed (at the start the phase will be

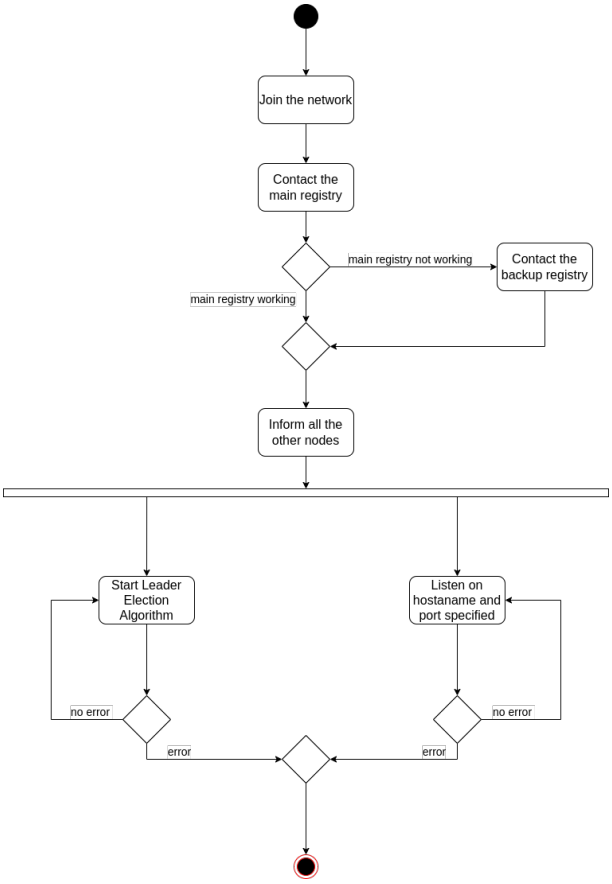

Fig. 2. General activity diagram of the node, it's the same for the two algorithms

0, and it is incremented each time the message reaches another node).
- The node candidate to become the leader.
- The sender of the election message.

If the next node does not respond (which means that the `DialHTTP()` or the `Call()` fail), the node will pass to the node after the next node in the ring, and so on until it reaches a working process to send the election message. This message is sent by calling the remote procedure `ElectionLeaderCR()` of the next node identified.

When the node receives an election message it checks if the ID is equal, smaller, or bigger than its. Depending on the case the behavior is:

- ID smaller: the election message is dropped. The node starts another election by invoking `ElectionCR()`.
- ID bigger: the node forwards the message to the next node. The function used to do this operation is the same as the one used to start an election (`ElectionCR()`).
- ID equal: that means that one election message sent in the past by the node has not been disregarded (which means no process in the network has an ID higher than its). The node becomes the leader and sets the flag `iamLeader` (the goal of this flag is to signal to the routine in the background that there is no need to check if the leader is alive because the leader is the node itself). Using the function `NotifyLeader()`, the node announces to every other process that it is the leader. The leader message is sent by calling the remote procedure `NewLeader()`.

Let's now examine the liveness and safety:

- Safety: "By the circular nature of the configuration and the consistent direction of messages, any message must meet all other processes before it comes back to its initiator. Only one message, that with the highest number, will not encounter a higher number on its way around. Thus, the only process getting its own message back is the one with the highest number" (citation of the original Chang-Roberts article [1]).
- Liveness: at least one node becomes the leader (all the possible failure scenarios are discussed in the RESULT section).

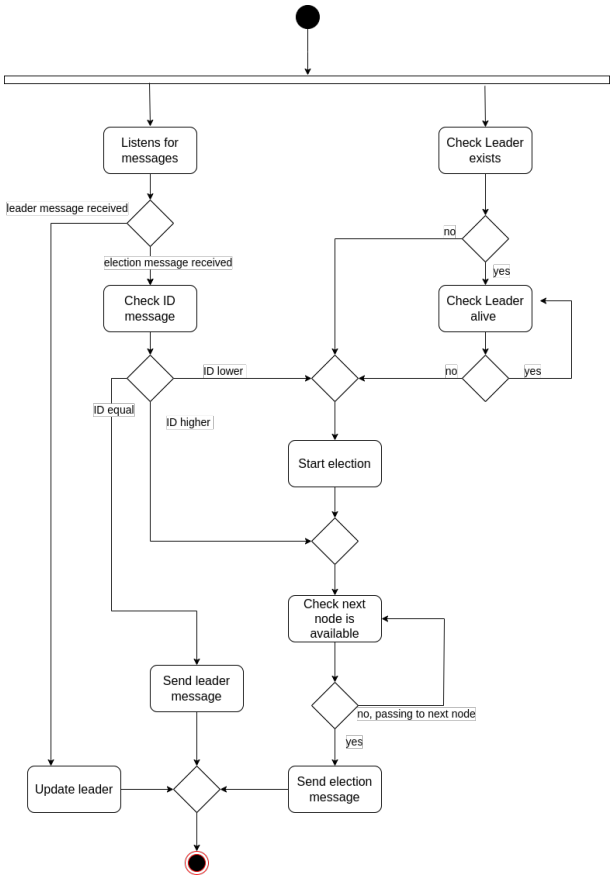

Fig. 3. Actvity diagram of the Chang-Roberts election

D. Election in Bully Algorithm

The function responsible for the election is `ElectionBully()`. At first, it sets the `election` flag and checks if the node is the only one in the system. If not the function scans all the nodes in the network searching for the ones with an ID higher than the ID of the node. Once

these processes are identified, the node checks iteratively if they are available and if so sends the election message by calling the remote procedure `ElectionBully()` for each one of them. If at least one of them responds then the node does not continue the election. If no node responds that means that there aren't any processes with higher IDs or they are not working. So the node becomes the leader and forwards a leader message to all the other processes in the network by calling `NotifyLeader()`. It also sets the flag `iamLeader`.
When a node receives an election message from a process with a lower ID then it responds back to it and starts a new election by calling `ElectionBully()`. The election message is the same as the one of the previous algorithm. So the two algorithms are equal in terms of the message's size.

Let's now examine the liveness and safety:

- Safety: if two processes start an election simultaneously when the process with a bigger ID is reached by the election message of the other node, it will respond to the latter (making it stop the election).
- Liveness: at least one node becomes the leader (all the possible failure scenarios are discussed in the RESULT section).

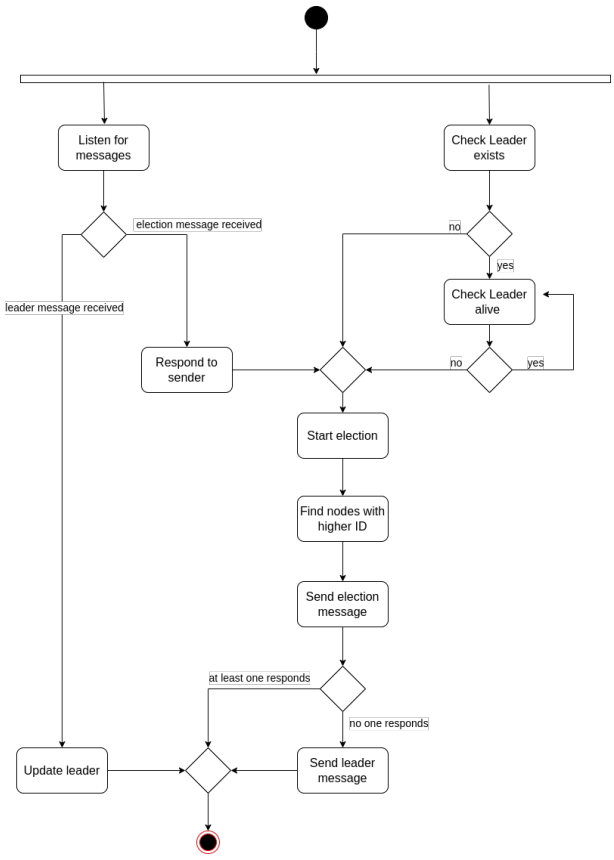

Fig. 4. Actvity diagram of Bully election

E. Rejoining the network

If a node fails and then starts again it can rejoin the network. What it does is exactly the same as entering the network the first time. However, the registry knows that the node has already been in the network by looking at its address. So the service registry will retrieve the ID of the node and send it back to the process, along with the list of peers in the network and the algorithm chosen.

IV. Results

The implementation has been tested in different scenarios of failure with a positive outcome for each of them. To simulate a node failure the corresponding container was stopped using the command `docker-compose stop <container>`. The number of nodes in the network during the tests was initialized at 15. Here is a list of possible failures:

- Failure of the leader after sending leader messages: at least one of the nodes in the network will detect the failure and start an election.
- Failure of the leader before sending leader messages: the calls to the function `NotifyLeader()` is linked to the procedure `ElectionLeaderCR()` (or `ElectionBully()`). So if the designated leader fails, the RPC to one of the two functions mentioned above will fail. Thus, the sender of the election message will consider the node as not working.
- Failure of one node while processing an election message: the RPC will return an error and the sender of the election message will consider the node as not working.
- Failure of a node that is not the leader: since the node is not the leader it is not a big issue. When one process wants to communicate with the failed node it will simply acknowledge that the latter is not working.
- Failure of the main registry while adding the node: the node will recognize that the main registry is not working (because the RPC `AddNode()` will fail) and it will contact the backup registry.
- Failure of the main registry while syncing with the backup registry: the function `GetPeers()` invoked by the new node will fail and the new node will not enter the network.

V. Discussion

In this section, some of the weaknesses of the system will be discussed:

A. Service registry as bottleneck

The registry, whether the main or the backup, can act as a bottleneck. In fact, there are two replicas of the registry but only for replication goals and not for load-balancing (only one registry at a time interacts with the nodes). One improvement of this implementation could be to use the two replicas also for load-balancing.

B. Maintenance of ring topology

The implementation expects that the nodes will contact the other nodes in the network to communicate their presence. Even though the testing does not report any problems in this scheme, the following scenario would be possible:
Node P_i wants to join and contact the registry. The registry gives to it all the information and the list of the nodes in the network and adds P_i to the peers. Let's suppose that the first node in this list is P_1 and the last is P_n. So P_i position in the ring is after P_n.
Then the registry is contacted by P_j. The registry sends again all the information and adds the node to the peers. P_j now comes after P_i. After receiving the lists of nodes in the network P_i and P_j starts sending messages to the other nodes following the order of such list. Due to latency or asymmetrical network, it *could* be possible that one node, for example P_1, receives the message of P_j before the message of P_i. So P_1 maintains *locally* a topology that is not coherent with the actual topology. The worst that can happen is that the node that should become the leader is gotten over by an election message.
Since the deployment of the system is done on a single host (so there are no problems due to latency), the implementation trades the low probability of the previous scenario with the reduction of the load on the service registry. If the deployment is made among different hosts then this problem can occur. To solve it, it's the registry that should inform all the other nodes of the joining of a new process.

C. Variable IP

The implementation expects that the nodes maintain the same IP address and port number when reconnecting to the network. In this way, the registry or the backup registry can disclose if the node has already been in the system or not. An improvement could be to distinguish the nodes using a particular identifier (such as the MAC address) allowing the processes to change their network addresses during their lifetime.

REFERENCES

[1] E. Chang, R. Roberts, An improved algorithm for decentralized extrema-finding in circular configurations of processes, Comm. ACM 22 (5) (1979) 281–283

[2] H. Garcia-Molina, Elections in Distributed Computing System, IEEE Transaction Comput, Vol.C-31, pp.48- 59, Jan. 1982