P1 Leader Election

Project report

A Distributed Systems Report

by Group 2:

Andreas Kaag Thomsen, 202105844

Buster Salomon Rasmussen, 202107215

Supervisors: Christian Fischer Pedersen & Christian Marius Lillelund

AARHUS UNIVERSITET

November 8, 2023

Contents

1 Introduction

Leader election in distributed computing designates a single node as the coordinator[1] among network nodes. Before working on a task, the nodes do not know which node will act as the leader. However, after the convergence of a leader election algorithm, all nodes are aware of the identity of the coordinator/leader. Efficient leader election algorithms are key to seamless task organization and network stability. This report provides a comparative analysis of the Bully Election Algorithm, focusing on the original version by Garcia-Molina and an improved variant inspired by more recent research. We explore their design, implementation, testing, and message complexity, aiming to showcase the efficiency enhancements achieved in the improved algorithm.

2 Methods and tools

In this project the algorithms have been implemented in object oriented Python, where each process have been implemented as a class running a thread. We have used Visual Studio Code as our integrated development environment (IDE) and Github for code sharing and version control. In the design phase we have used the unified modelling language (UML) to showcase the design and architecture of our implementation.

3 Results

In this section, we present the design and implementation of both the original and improved bully election algorithms, focusing on algorithm details and enhancements in message complexity. The latter is elaborated in the Testing section, where we delve into unit tests and system tests.

3.1 Original Bully Election Algorithm

The original Bully Election Algorithm was first presented by Garcia-Molina [1], and our implementation is very similar to the one presented here. The original bully election algorithm is an intuitive leader election algorithm, based on the idea that higher ID processes bully lower ID processes to drop out of the election [1]. In general our algorithm implementation is defined as follows:

Let the processes be defined $\{P_i\}_{i=0}^{N-1}$. *Then* P_k *initiates an election due to a non-corresponding coordinator:*

1. P_k sends an election message to all processes with higher IDs than its own (state: ELECTION)
2. P_k awaits OK messages (state: WAITING_FOR_OK)
   - If P_k does not receive any OKs or time T passes, then it becomes coordinator, and sends out coordinator messages to processes with lower IDs (state: COORDINATOR)
     - When a process receives this message it treats the sender as the new coordinator (state: NORMAL)
   - Otherwise P_k drop out of the election, and the process(es) P_i where $i > k$ takes over and start(s) their own elections from step 1.

Note that the algorithm will cause parallel elections since the processes do not wait for one election to finish before starting its own. That is, a new election is started immediately after step 1

[1] In this report the terms 'leader' and 'coordinator' are used interchangeably.

3.1.1 Design

In this section, we elaborate on the design of the original bully election algorithm. This is showcased using UML diagrams.

3.1.1.1 Class diagram The class diagram depicted in Figure 1 shows the attributes and methods used to implement the bully algorithm. Every attribute will not be described in detail here since many are trivial, but among the most crucial to the workings are:

- `state_machine` which is the main loop, a thread, of each process with respect to the bully algorithm. The state machine diagram, which illustrates the states each process can take, orchestrated by this method, is visualized and explained in detail in the Bully Original State Diagram section
- `message_queue` which a queue for incoming messages for each process.
- `message_handler` which is a helper function invoked by `state_machine` for each incoming message, and in turn takes the appropriate action based on the `msg_type`.

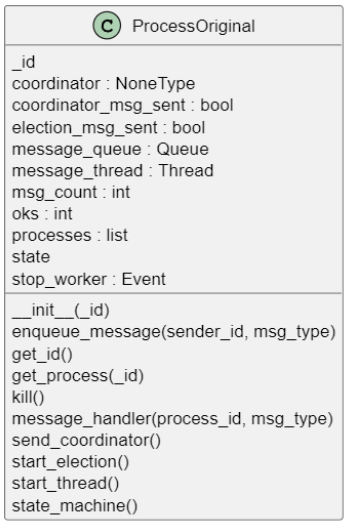

Figure 1: Process Class Diagram For Original Bully Algorithm

In summary, the `message_queue` contains messages from other processes and this is emptied within the method `state_machine`. If it is empty (i.e. there are no messages) the state machine logic is executed. If there is a message, this is handled by the method `message_handler`.

3.1.1.2 Bully Original State Diagram In our implementation, each process can be in one of four states as depicted in the process state diagram in Figure 2. Each process starts in `NORMAL`. From `NORMAL` it can either be killed[2], and go to `DEAD`, from where it cannot return[3], or start an election and go to `WAITING_FOR_OK`. If it does receive any OKs within a given time threshold, it will go to `WAITING_FOR_COORDINATOR`, and if it does not receive any OKs it will go to `COORDINATOR`. From here it

[2]The terminology of a process being killed is used to denote the simulation of a process failure.
[3]Because this is merely a simulation it is not possible to get revived upon being killed.

can only go to `DEAD` if being killed.

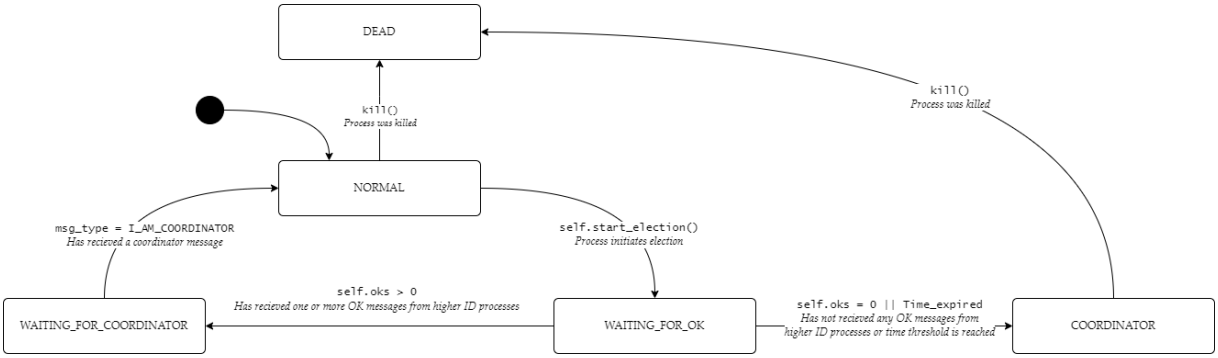

Figure 2: UML State Diagram for the original bully algorithm

3.1.2 Implementation

The core logic of the Bully Election algorithm is defined by the state the current process is in. For that reason, the method `state_machine`, as shown in Listing 1, implements the main features of the algorithm.

```
def state_machine(self):
    while not self.stop_worker.is_set():
        try:
            msg_type, process_id = self.message_queue.get(timeout=1)
        except Empty:
            if self.state == NORMAL or self.state == WAITING_FOR_COORDINATOR:
                pass
            elif self.state == COORDINATOR:
                if not self.coordinator_msg_sent:
                    self.send_coordinator()
            elif self.state == WAITING_FOR_OK:
                time_passed = time.time() - self.election_start_time
                time_expired = time_passed > THRESHOLD
                if self.oks > 0:
                    self.oks = 0
                    self.state = WAITING_FOR_COORDINATOR
                elif time_expired:
                    self.send_coordinator()
                else:
                    pass
        else:
            self.message_handler(process_id, msg_type)
```

Listing 1: Implementation of the state machine in the Original Bully Election Algorithm

This method operates within a dedicated thread, continuously monitoring incoming messages from other processes while executing the state machine logic as shown in Figure 2. Specifically, when the process is in the `WAITING_FOR_OK` state (line 11), indicative of an election initiation, it checks for the arrival of OK messages. Upon receiving OK messages, the process abandons its bid for coordinatorship and transitions to a waiting state for a coordinator message as shown in line 16. If no OKs arrive within a given time threshold, the process elects itself as coordinator. If any message arrives, it is handled by the `message_handler` method in line 22, a straightforward process not detailed here. Essentially, receiving `OK` messages increments the OK count, `I_AM_COORDINATOR` messages sets the process' state to `NORMAL` and updates the current coordinator. When process i receives an `ELECTION` message from process j, it responds to j with an `OK` message and commences its own election. This mechanism triggers a cascade of elections, contributing to the inherent inefficiency of the original Bully Election Algorithm.

3.2 Improved Bully Election Algorithm

As an improved version of the bully algorithm we have found inspiration from the paper found in [4]. This paper states that "the two major limitations of Bully algorithm are the number of stages to decide the new leader and the huge number of messages exchanged due to the broadcasting of election and OK messages" [4, p. 47].

3.2.1 Design

In this algorithm two assumptions have been introduced for each process, namely that each process has an election flag and a coordinator field for storing the `id` of the believed coordinator [4]. These are the only things changed in the class diagram and are therefore not shown here. The most prominent way this algorithm differs from the original is that only one process is allowed to run an election at a time[4]. When the processes receives election messages, they respond with an OK and their id, and then the elector finds the processes with the highest ID from these responses and notify the corresponding process that has the highest ID. Then that process makes a crosscheck by starting an election, and if it receives no responses, it becomes the coordinator and notifies the rest of the system. Thus, the chain of elections is avoided. Based on this, the state diagram for the improved algorithm is equivalent the the one for the original shown in Figure 2.

3.2.2 Implementation

The implementation is somewhat similar to the original implementation with only slight differences. A new message type has been added `YOU_ARE_COORDINATOR`, to indicate that a process has been elected coordinator by the elector. This leads to some notable changes in the state machine's branch `WAITING_FOR_OK`, depicted in Listing 2. The first check at line 4, has the same functionality as in the original. If it is the case, there are two additional cases. 1) Line 5, No OKs have been received from the crosscheck, which indicates that the election was started by the highest ID process 2) Line 8, OKs have been received, which implies that a `YOU_ARE_COORDINATOR` message should be sent to the highest ID process (which will start a new election and end up in case 1 [5]).

[4] Using the election flag
[5] Unless there are higher ID processes, that are not available to the initial elector

```
elif self.state == WAITING_FOR_OK:
    time_passed = time.time() - self.election_start_time
    time_expired = time_passed > THRESHOLD
    if self.oks == len(self.processes)-self._id+1 or time_expired:
        if self.oks == 0:
            self.current_coordinator = self._id
            self.state = COORDINATOR
        else:
            new_coordinator = self.get_process(
                self.current_coordinator)
            new_coordinator.enqueue_message(
                self._id, YOU_ARE_COORDINATOR)
            self.state = WAITING_FOR_COORDINATOR
        self.oks = 0
```

Listing 2: Notable changes in the implementation of the state machine in the Improved Bully Election Algorithm

3.3 Testing

For the testing we have performed both unit testing as well as system testing. For both parts we have utilized the '`unittest`' unit testing framework from the Python standard library.

3.3.1 Unit testing

We have written unit tests for almost all the methods except for the methods `start_election()`, `state_machine`, and `message_handler`. These three are tested simultaneously when testing the entire system. This will be elaborated in subsubsection 3.3.2. Using the aforementioned test framework, we have used the built-in method `setUp` for setting up the test fixture before exercising it. Here, we create N processes and start the thread within each process. Each test case tests exactly one method and uses an `assert`-method to check the expected outcome. As an example, the test case for the method `enqueue_message` is shown in Listing 3.

```
def test_enqueue_message(self):
    for i in range(self.N):
        process = self.all_processes[i]
        process.enqueue_message(i, "TYPE")
        self.assertEqual(process.message_queue.get(), ("TYPE", i))
```

Listing 3: Test case for method '`enqueue_message`'

The expected outcome of this function is that the `message_queue` contains the enqueued object. This is verified using the `assertEqual`-method. The rest of the test cases are constructed in a similar manner for both the original and improved bully election algorithm. Running all 12 test cases from the terminal gives the result:

```
python unit_test.py
............
----------------------------------------------------------------------
Ran 12 tests in 0.022s
OK
```

indicating that all tests pass for both algorithms.

3.3.2 System testing

The system testing focuses on whether or not the implementations converge. Here, convergence is defined as all processes/nodes having the same coordinator and the coordinator being the node with highest ID after a period, T, where T can be an arbitrary amount of time. Again, we have made use of the `setUp` method as well as the `tearDown` method from the testing framework. In the `tearDown` method we also calculate the total number of messages sent, which is the metric for the performance. The test case for `start_election` is shown in Listing 4. The test case for the original algorithm is very similar to this.

```
1  def test_election(self):
2      self.all_processes[0].start_election()
3      sleep(10)
4      for i in range(self.N-2):
5          self.assertEqual(self.all_processes[i].state, NORMAL)
6      self.assertEqual(self.all_processes[self.N-1].state, COORDINATOR)
7      for i in range(self.N-1):
8          self.assertEqual(
9              self.all_processes[i].current_coordinator, self.N-1)
```

Listing 4: System test case for `start_election` in the improved bully election algorithm.

Here, we wait for $T = 10s$ after node 0 has initiated the election. Next, we check that the processes have the correct states. That is, the process with highest ID $(N-1)$ should be in state $COORDINATOR$ whereas the remaining nodes should be in state $NORMAL$. In addition, all nodes should agree upon the coordinator being the node with highest ID.
Executing the system tests yield:

```
python system_test.py
Total messages sent in improved: 26
.Total messages sent in improved: 13
.Total messages sent in original: 44
.Total messages sent in original: 24
.
----------------------------------------------------------------------
Ran 4 tests in 59.014s
OK
```

Note how we have only shown the scenario corresponding to initial startup of the system. We have also tested the scenario that the coordinator node fails and another node sees this. These test cases also pass.

4 Discussion

This section analyzes and compares the message complexity of the implemented Bully Election algorithms. We explore their computational performance and discuss potential improvements in leader election algorithms for distributed systems.

4.1 Quantitative comparison

For the quantitative analysis, we have used the results from the system testing. From this we have obtained the number of messages exchanged in the system when the lowest ID node initiates an election. The results are depicted in Figure 3.

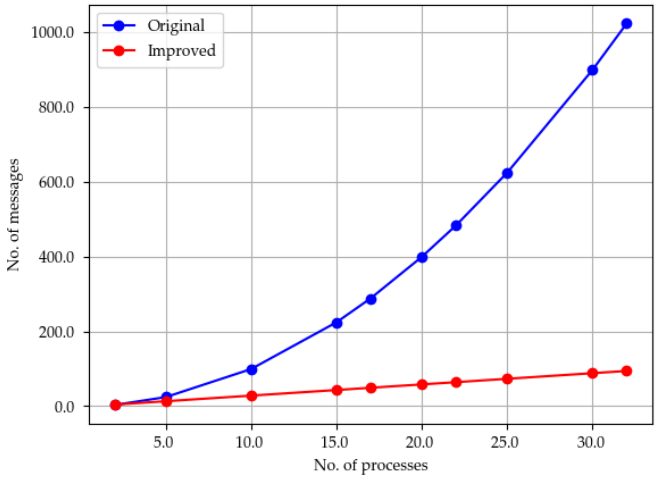

Figure 3: Number of messages exchanged in the system during an election

The graph visually demonstrates the increased efficiency of the improved bully election algorithm, notably in terms of message reduction. The comparison reveals a quadratic growth in the number of messages for the original algorithm with respect to the number of processes, while the improved algorithm exhibits a linear growth. Further elaboration on this comparison in terms of big-oh notation is provided in the following section.
In addition, we have also performed quantitative analysis using print statements[6]. These revealed that there are no parallel elections when using the improved algorithm also contributing to the enhanced efficiency of the algorithm. Another important aspect is that both algorithms are reliable and will always converge on a coordinator given the assumptions elaborated in detail in [1].

4.2 Analytical comparison

In this section, we delve into the complexity of the two implementation. More specifically, we investigate the worst-case message complexity in terms of the number of nodes and the node that initiates the election.

[6]These are now removed from the final code.

4.2.1 Original Bully Election Algorithm

Considering N processes with IDs from $\{1 \dots N\}$[7] where process k initiates the election. In the original algorithm $P(k)$ will send out election message to $N-k$ processes, $P(1)$ to $N-(k+1)$ processes and so on. Next, each process will receive the same number of OK messages. Finally, $P(N)$ will inform $N-1$ processes of its new status as coordinator. The exact number of messages sent is given by Equation 1

$$2\sum_{i=k}^{N}(N-i)+(N-1)=(k-N-1)(k-N)+N-1 \tag{1}$$

Considering the worst case message complexity in which $k=1$ (lowest priority process), Equation 1 reduces to N^2-1 from which the message complexity can be deduced as $O(N^2)$.

4.2.2 Improved Bully Election Algorithm

Once again we consider N processes with IDs from $\{1 \dots N\}$ and process k initiating the election. In the improved algorithm, process k will send $N-k$ election messages and receive the same amount of OK messages. Finally, $P(N)$ will send $N-1$ coordinator messages. Hence, the exact number of messages sent is given by Equation 2

$$(N-k)+(N-k)+(N-1)+1=3N-2k \tag{2}$$

Considering the worst case in which $k=1$, we deduce the message complexity to be $O(N)$. From this it is clear that the improved Bully Election Algorithm performs better since the worst-case message complexity is reduced from its quadratic nature to linear, which was also vivid in Figure 3.

4.3 Perspectives

Many other approaches of leader election exist, and two of the most widely used are RAFT and Paxos [2]. These are both consensus algorithms, which have leader election as an integrated part of the algorithm. RAFT is from 2014 and is often used due to its lightweight leader election and simplicity [2]. In this section we delve into the design of RAFT and a short comparison with Molina's algorithm.

4.3.1 Design of leader election algorithm in raft

The RAFT leader election process takes on the same structure as a representative democracy. That is, it is based on votes and the candidate with the most votes wins the election. Thus, the election outcome is independent of process IDs. This is described in detail below based on the RAFT paper in [3].
After a process has realized that a leader is down, an election is initiated after a randomized election timeout, to decrease the risk of split votes. This is done by incrementing its term, transitioning to candidate state, voting for itself and sending vote requests messages to the rest of the processes. If the receivers have not already voted for another candidate, it accepts the request and votes for requester (first-come-first-serve basis). If a candidate itself receives a vote request, it accepts it and transitions to follower state, if the term is at least as large as its own. The process with the most candidate wins[8].

[7] In this analysis we consider - without loss of generality - 1-indexed processes as opposed to the implementation that uses 0-indexed notation. This is for simplicity in the obtained formulas.

[8] It is not explicitly stated in section 5.2 of the article when and how this evaluation is performed

4.3.2 Comparison

The performance of the leader election algorithm in RAFT is not given in terms of message complexity, but time to find a new leader. This is possibly done because the message complexity is not deterministic due to the randomness in the algorithm. For that reason, we will not compare the algorithm with respect to performance metrics, but rather give a more general analysis for why this leader election algorithm can be preferable.

According to Molina [1], the original bully election algorithm is not safe[9] if the system does not guarantee instantaneous responses and no communication subsystem failures. However, in many practical applications such guarantees cannot be complied with. Since RAFT includes the notion of clusters - similar to groups in the invitation algorithm [1, p. 53] - safety *can* be guaranteed, which is a major factor in the selection of a leader election algorithm. The notion of clusters also means that RAFT does not require full network topology, further reinforcing its position as one of the most widely adopted leader election algorithms in modern systems with numerous nodes.

5 Conclusion

In this report, we implemented, verified and compared two versions of the Bully Leader Election Algorithm: the original and an improved version. The design of both algorithms were showcased using UML diagrams and they were both implemented and tested in Python. The experimental phase confirmed the successful leader election functionality of both algorithms, with the improved version notably showcasing superior efficiency in terms of message complexity. The result of our analytical comparison showed that the improved algorithm reduced the worst-case message complexity from $O(N^2)$ to $O(N)$ with N being the number of processes in the system. Additionally, we discussed perspectives in the context of the newer consensus algorithm RAFT. RAFT is independent of IDs and is based on groups, thus not requiring full network topology. RAFT is considered state of the art and is widely used due its lightweight design and simplicity.

6 References

[1] Garcia-Molina. "Elections in a Distributed Computing System". In: *IEEE Transactions on Computers* C-31.1 (1982), pp. 48–59. DOI: 10.1109/TC.1982.1675885.

[2] Heidi Howard and Richard Mortier. "Paxos vs Raft". In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Apr. 2020. DOI: 10.1145/3380787.3393681. URL: https://doi.org/10.1145%2F3380787.3393681.

[3] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[4] P Beaulah Soundarabai et al. *Improved Bully Election Algorithm for Distributed Systems*. 2014. arXiv: 1403.3255 [cs.DC].

[9] More than one leader can be elected