Leader Election - Improving the Bully Algorithm
Group 1.

Liulihan Kuang
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201906612@post.au.dk

Jeppe Stjernholm Schildt
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201905520@post.au.dk

Kristoffer Plagborg Bak Sørensen
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201908140@post.au.dk

Alexander Stæhr Johansen
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201905865@post.au.dk

***Abstract*–This report fulfils the purpose of documenting the design, implementation and verification of the regular Bully Algorithm developed by Héctor García-Molina and a modified version. The algorithms have been implemented successfully in C++, supplemented by an almost successful Python version as well. Both implementations have been visualized formally applying UML in the framework of Kruchten's "4+1" model, with specific usage of the "logical view" and the "deployment view", and with testable code, realized as scenario simulations, which can be found in the associated repository [5]. The success of the two algorithms have been concluded based on scenario testing and performance testing, and doing so proved the time complexity of the regular Bully Algorithm being quadratic and the modified version being linear. Lastly, a parallel to state of the art leader election algorithms have been made.**

Definitions

Word	Description
UML	Unified Model Language
PID	Process Identification

Introduction

The following report has been realized to document a project in the course "Distributed Systems" at Aarhus University as part of the bachelor's degree in Computer Engineering on the 3rd semester.

I. Context, project purpose and objectives

In the following section, the context of the project will be outlined, followed by the purpose of the project and finally which objectives this project aims to achieve.

Context

To understand what the Leader Election problem entails, one must first understand what a distributed system is. Albeit it can be defined in a plethora of ways one suitable, for this project, is that a distributed system consists of multiple computers connected via. a network that interact with one another in order to achieve a common goal [6]. This common goal is organized by one computer, also called a "node" or "process", on the network, and this organizer is named the "coordinator". If the coordinating process fails, a new coordinator must be elected. This problem is named the "Leader Election problem". In distributed systems, one is interested in assuming an autonomous leader election process and one such solution is the "Bully Algorithm".

Project purpose

The purpose of this project is to design, implement, verify, compare and document:

- The regular Bully Algorithm
- An improved version of the Bully Algorithm

Objectives

Within the purpose presented above, one may divide the objectives into two dimensions, one qualitative and one quantitative, presented as below.

Qualitative:

- Design and implement the regular Bully Algorithm and an improved Bully Algorithm in Python
- Using UML, visualize the design and implementation of the two algorithms
- Verify the functionality of the two algorithms
- Compare and discuss the two algorithms quantitatively

Quantitative objectives:

- Document message and time complexity of the two algorithms
- Verify the superiority of the modified Bully Algorithm with prioritized respect to message complexity and time complexity numerically

II. SPECIFICATIONS

In the following section the regular Bully Algorithm will be explained and its pitfalls outlined, followed by how these pitfalls are mitigated by the modified Bully Algorithm.

Regular Bully Algorithm

The regular Bully Algorithm was initially introduced by Hector Garcia-Molina in 1982. The regular Bully Algorithm is used for dynamically electing a coordinator process in a distributed system, in which the current coordinator process has failed. The regular Bully Algorithm operates by electing the process with the highest ID to be the coordinator.

To understand the regular Bully Algorithm, one must first become familiar with its underlying assumptions, paraphrased below for applicability [2, page 49-50.]:

- Assumption 1: Each process has a unique ID.
- Assumption 2: Processes know each other's ID.
- Assumption 3: Each process can compare IDs.
- Assumption 4: Processes do not know which ones are currently alive.
- Assumption 5: All processes can intercommunicate.
- Assumption 6: A failed process is always detectable.
- Assumption 7: Any process can initiate an election.
- Assumption 8: Upon failure recovery, the process knows it has failed.
- Assumption 9: Message delivery between processes is reliable and time bound.
- Assumption 10: All processes follow the same election algorithm.

Upon these 10 assumptions, the algorithm is founded and it can be explained in 3 basic steps [4, page 45.]:

1) **Step 1:** Election initiation
   When a process P becomes aware that the coordinator has failed, it initiates the election algorithm. Then:
   a) P sends an election message to all processes with higher IDs than itself.
   b) If no one responds within the fixed time limit T, then P wins the election and becomes the coordinating process.
2) **Step 2:** Receipt of election message
   When a process receives an election message from one of the processes with a lower PID than itself, than it:
   a) Sends an OK message back to the sender to notify that it is alive and now in charge.
   b) The receiver then initiates an election, unless one is already being executed.
   c) All processes terminate the election except the one with the highest PID, which is now the coordinator.
   d) The new coordinator announces its victory by sending a message to all processes and telling them that it is leading.
3) **Step 3:** Higher PID restoration
   Immediately after a process gets reactivated and rejoins the network, it initiates an election as it has no way of telling way of knowing which processes are currently up (assumption 4).

To fully grasp the algorithm, an example scenario has been visualized, which can be found in the Github [5],

named "Example scenario, regular Bully Algorithm". The figure depicts the following steps:

A Process 5. has failed, and process 2. has noticed. Therefore, process 2. sends an election message to all processes with greater IDs than itself.

B Process 3. and 4. tells process 2. OK, making process 2. stop its election, and initiate their own elections.

C Process 3. and process 4. initiates an election, however, in the figure only the election of process 3. is shown.

D Process 4. tells process 3. OK, making process 3. stop it's election.

E As no processes with higher PID than process 4. responds within the time T, process 4. wins the election and becomes the coordinator. This victory is announced to all processes.

Advantages and disadvantages of the regular Bully Algorithm

Some advantages of the regular Bully Algorithm are that it is easy to implement and it ensures that the coordinating process is alive, assuming message reliability. The main drawback of the regular Bully Algorithm is that it imposes a very large amount of traffic to the network upon which it is connected. The reason for this is to be found within the number of messages the algorithm imposes. If one was to calculate the exact number of messages being passed in the worst-case scenario, one would need to sum the number of election messages, the number of OK messages and the number of victory messages sent on the network. The number of election messages sent by one process must be equal to $P - p$ where P is the total number of processes on the network and p is the process noticing the collapsed coordinator. This difference should be calculated for every process following the one that notices the crashed coordinator, hence the total number of election messages is:

$$M_{Election} = \sum_{p_0}^{P=p} (P - p) \tag{1}$$

The number of OK messages can be calculated in a similar manner, however, as one process is collapsed it cannot respond with an OK message, this sum must be given by:

$$M_{OK} = \sum_{p_0}^{P=p} (P - 1 - p) \tag{2}$$

Finally, the number of victory messages must be given by:

$$M_{Victory} = P - 1 \tag{3}$$

Therefore the total number of messages sent in a worst-case scenario is given by:

$$M_{Total} = \sum_{p_0}^{P=p} (P - p) + \sum_{p_0}^{P=p} (P - 1 - p) + P - 1 \tag{4}$$

Which means the number of election and OK messages has a complexity of $O(P^2)$ while the number of victory messages has one of $O(P)$, resulting in an overall time complexity of:

$$O(P^2) \tag{5}$$

The modified Bully Algorithm

In the previous subsection, it was declared that the primary disadvantage to the regular Bully Algorithm is that of message complexity. Exactly this is the motivation for the modified Bully Algorithm.

The assumptions previously stated should still hold, however, this time two additional are added [4, page 47.]:

- Assumption 11: All processes have an election flag, and if this is active then another election cannot be held.
- Assumption 12: All processes have a variable to store information about the current coordinator.

Having now implemented an additional two steps, one can outline the process of the modified Bully Algorithm applying a step-wise approach, similar to that given for the regular Bully Algorithm:

1) **Step 1:** Election initiation
   To begin with, all election flags associated with all processes are set to false. When a process P observes a crashed coordinator, it begins the election, by:
   a) Sending an election message to all processes.
   b) All processes set their election flag to true, so no other election can be facilitated.
   c) Coordinator variable is set to zero.
   d) If none responds P wins the election and becomes the coordinator.

2) **Step 2:** Election message receipt. When a process P receives an election message from another process Q with a lower PID, the process:
   a) Sends an OK message message back to the sender, indicating that it is active and takes over the election.
   b) The sender retrieves the process ID of the receiver and stores it in its coordinator variable, if the process ID of the receiver is greater than the one currently stored.
   c) All processes then respond and the highest ID among them is stored in the coordinator variable.
   d) The elected coordinator process checks if the processes greater than itself are alive and if not, it will elect itself as the coordinator.
   e) The coordinator then announces its victory by sending a victory message to all processes.
   f) All processes extract the PID of the new coordinator and saves it in their coordinator variable and sets the election flag to false.
3) **Step 3:** Higher PID process restore. If/when the process with a PID greater than the current coordinator is alive again, the algorithm is rerun.

Again, to understand the modified Bully Algorithm, an example scenario of its behavior has been outlined stepwise below and visualized in an example scenario, which can be found in Github [5]) in the appendix:

A Process 5. has failed, and process 2. has noticed. The first thing it does is to send a halt message over the network to all the other processes, signalling them not to start an election, when they happen to notice as well that the coordinator is down. Then process 2. sends an election message to all processes with greater PIDs than itself.

B Process 3. and 4. tells process 2. OK, making process 2. stop its election, and initiate their own elections. This makes process 2. store the PID of the largest of the two, i.e process 4.

C Process 2. then tells process 4. that it should host its own election.

D Process 4. checks with process 5. if it is up yet.

E As process 5. is not up yet, process 4. informs all processes of its victory, and they should update their coordinator-id variable to be 5, and unhalt themselves.

The advantage of the modified Bully Algorithm

The advantage of the modified Bully Algorithm presented above is its message complexity. The reason being that it only sends $(P-p)$ election messages, $(P-1-p)$ OK message and $P-1$ victory messages. Therefore, the total number of messages sent by the modified Bully Algorithm is:

$$\begin{aligned} T_m &= M_{Election} + M_{OK} + M_{Victory} \\ &= (P-p) + (P-1-p) + (P-1) \end{aligned} \tag{6}$$

Which results in a time complexity of:

$$O(P) \tag{7}$$

Which is indeed a significant reduction in message- and time complexity.

III. Implementation

The initial approach was to implement both the regular Bully Algorithm and the modified Bully Algorithm in Python, however, during this process, it was discovered that doing so would impose violations of assumption 9). This assumption states that communication is time bound and reliable, however, reliable communication can not be insured because of the Python "Global Interpreter Lock" [1]. This lock makes the thread schedule inconsistent with respect to time when process switching, meaning massive delays were experienced, hence the functionality of the algorithm corrupted.

Due to this error, it was decided to implement the algorithms in C++ instead and focus entirely on these in this report. The Python algorithms have been included in the repository [5], as they are considered a very valuable component of the design process, but they will not be further explained in the report. The reader is, however, strongly encouraged to investigate these, and the python implementation of the regular bully election algorithm is functional and working. But due the inconsistent scheduling of threads on multi-core computer processor architectures we found it difficult to in an reliable way getting each process to print the number of messages sent.

To create a visual overview of the implementation of both algorithms, it has been decided to use the "logical view" and the "deployment view" from the "4+1 view

model" by Kruchten [3]. These visualizations cannot work as a stand-alone foundation, so after having revisited the class and state diagrams, the reader is urged to dwell in the repository related to this project [5], in which comments are supplied to the code throughout.

The logical view

To visualize the main classes, their underlying attributes and methods, a class diagram for both the regular Bully- and the modified Bully Algorithm, has been realized and both can be found in the Github [5] appendix.

The deployment view

Within the regular Bully Algorithm, the most important deployment to consider, is that of which states a process goes through. To illustrate this, a state diagram as the following has been realized, which is relevant for both algorithms as the states are equal:

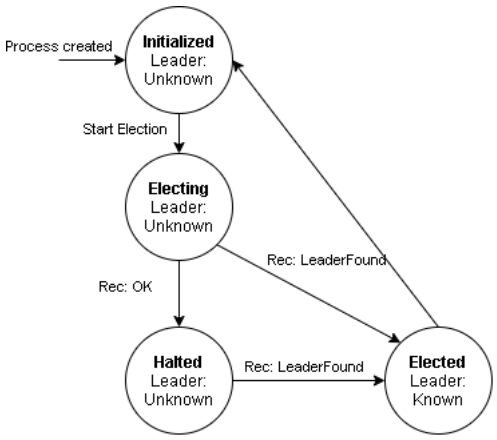

Fig. 1

When a process is initialized (new or recovered from a crash state), it automatically starts an election. The process that holds the election can either receive OK, from another process with a higher ID, or it can receive a notification message of an elected coordinator. In the first case, the process goes into a halt state, transferring the election holder status to a new process, and then waiting for the election to be finished but not participating any further itself. Whenever the election is over, the elected coordinator sends every other process a message notifying them of its victory, with information about itself, making them update their internal coordinator information.

IV. Verification and results

To verify the two algorithms, scenario testing, unit testing and analysis of message complexity has been done.

The scenario testing ensures that the algorithms execute according to their respective definition, paraphrased stepwise in the "specifications" section of this document. Therefore, an obvious test-strategy is to conjure a scenario for each step and verify the performance.

The unit testing ensures that individual parts/functions of the code which constitute the overall functionality of the algorithm is testable, so the applied test-strategy is determining whether fundamental invariants hold true. This has been done implementing a test-suite consisting of multiple assert statements. In this report, only the test suite of the modified Bully Algorithm is visualized, albeit the test suite of the regular Bully Algorithm can be observed in the code.

Lastly, both implementations will have their message complexity compared for different numbers of processes in the network, considering the worst-case scenario.

In the following three subsections, each of the algorithms will be verified and tested with respect to scenario- and unit testing. In the last subsection, the message complexity of the two algorithms will be analyzed.

Regular Bully Algorithm

First, the scenario testing of the regular Bully Algorithm:

Fig. 2: Deactivating a coordinator

When process 5 find out that the current coordinator is dead, it starts an election, so it sends a message to all the processes that have a higher ID than itself, if

they are alive, they will reply to process 5. The higher ID process do the same, sending messages to the even higher ID processes. There were 24 messages sent in total in this election.

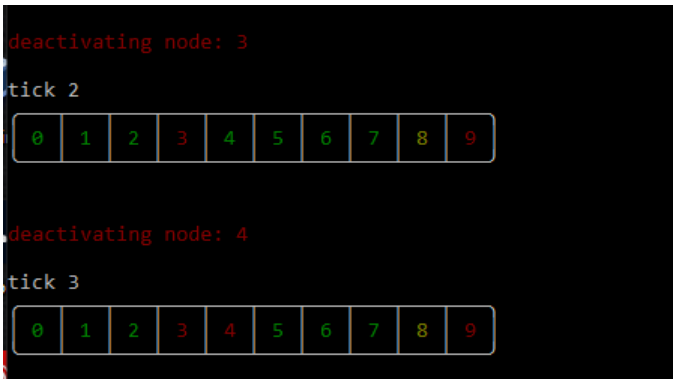

Fig. 3: Deactivating normal processes

When the normal processes are down, no messages are sent.

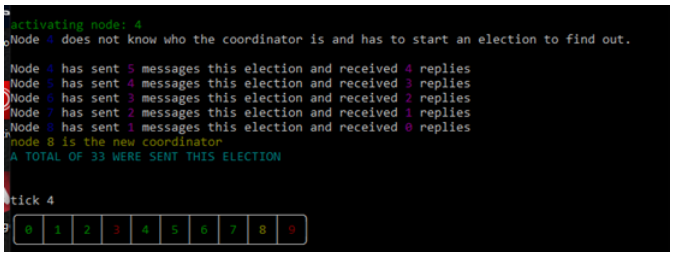

Fig. 4: Recovering a normal process

When a normal process has been recovered, it wants to know who the coordinator is, so it starts an election, sending messages to all processes bigger than itself.

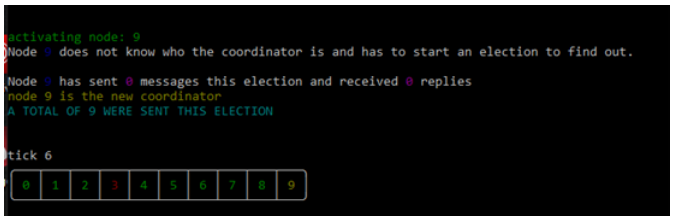

Fig. 5: Activating the process that have a higher PID than the current process

When the highest process has been activated, it will send no election message to others because there is no other process that has a higher PID than itself. It will just send victory messages to all the processes that has a lower PID than itself.

Modified Bully Algorithm

First, the scenario testing of the modified Bully Algorithm:

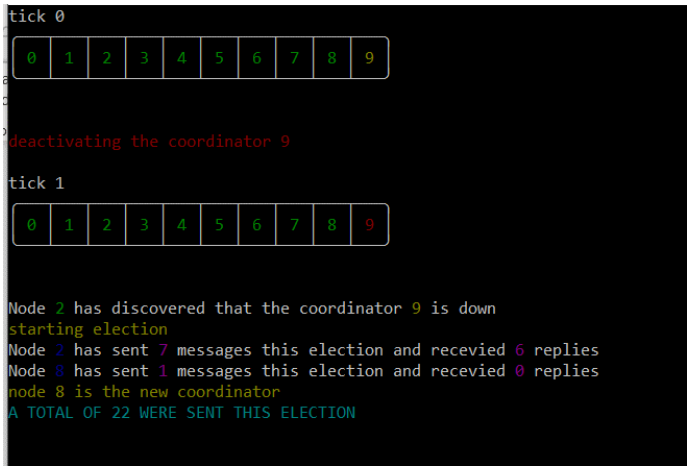

Fig. 6: Deactivating the coordinator

When process 2 finds out that the current process is dead, it starts an election by sending messages to all the processes that has a higher PID than itself. Then, the highest ID that replied ”process 2” tries to send messages to the even higher process (if there exist any, in this case process 8 is the highest process that replied process 2). Process 8 got no response from the higher process; thus, it is the new coordinator.

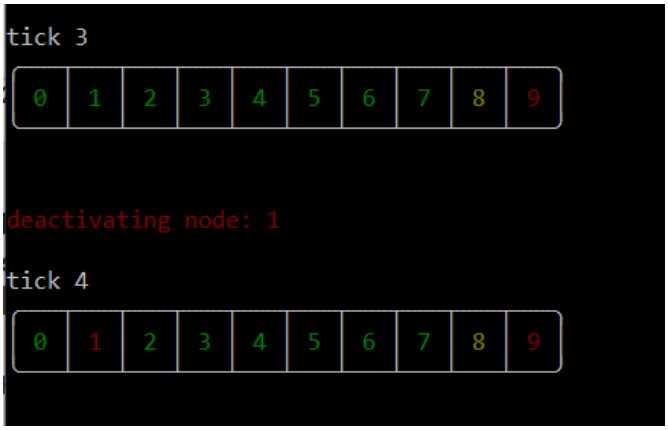

Fig. 7: Deactivating a normal process

When a non-coordinator process is been killed, nothing happens.

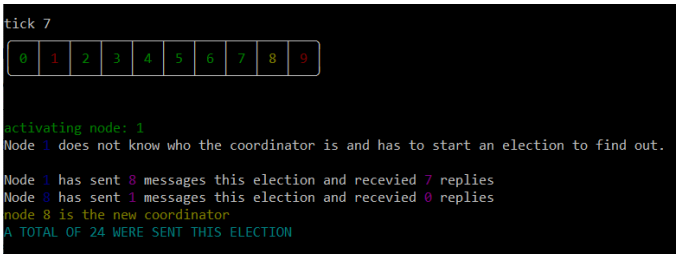

Fig. 8: Recovering a normal process

When a non-coordinator process is recovered, it doesn't know who is the current coordinator, therefore it starts an election by sending messages to all the processes that have a higher ID than itself, and the highest process that replies, takes over, sending a message to the even higher processes. Then, if no one replies, it process elects itself.

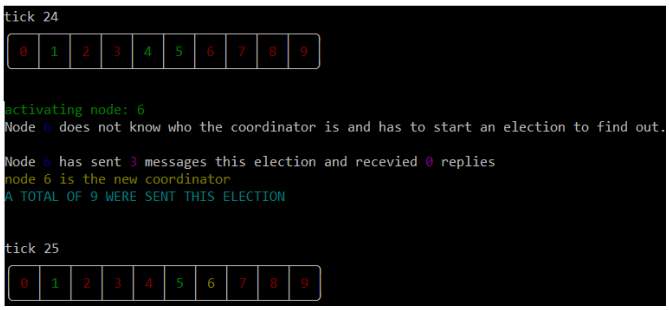

Fig. 9: Recovering a process that is higher than the current process

When recovering a process that has a higher ID than the current coordinator, the recovered process starts an election by sending messages to the processes that have a higher ID than itself, if no one replies, it becomes the new coordinator.

Second, the unit testing of the modified Bully Algorithm:

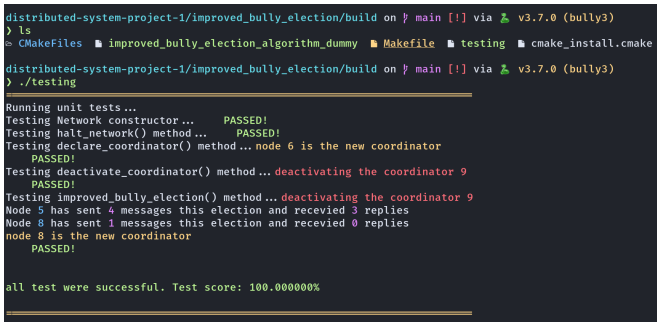

Fig. 10: Deactivating a coordinator

Message complexity comparison

In the table below, the number of messages in the worst-case scenario for a different network size N has been tested for both the regular and the modified Bully Algorithm. The columns annotated E represents the expected values, considering the formulas presented in the specifications section, while the R columns declare the realized number of messages. Documentation of these tests have not been included in the report, as the reader can replicate them easily, applying the developed simulations, supplied in the repository [5]:

	Messages			
	Regular		Modified	
Processes	E	R	E	R
4	11	11	8	8
6	29	29	14	14
10	89	89	26	26
14	181	181	38	38
18	305	305	50	50
22	461	461	62	62
24	551	551	68	68
28	775	775	80	80

Applying this data one can calculate the absolute and relative change in the number of messages sent in the previously applied scenarios:

	Algorithm		**Improvement**	
Processes	Regular	Modified	Absolute	Relative
4	11	8	3	-27,27%
6	29	14	15	-51,72%
10	89	26	63	-70,79%
14	181	38	143	-79,01%
18	305	50	255	-83,61%
22	461	62	399	-86,55%
24	551	68	483	-87,66%
28	775	80	695	-89,68%

Finally, one can map the number of processes, on the y-axis against the number of messages sent for both algorithms, on the x-axis, in a plot to compare the number of messages and derive a mathematical expression for the tendency lines:

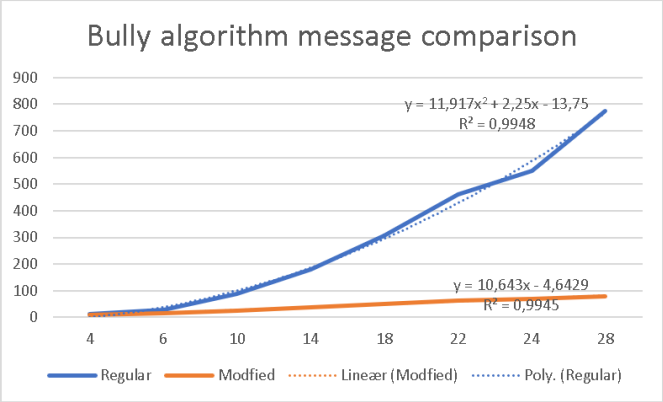

Fig. 11

V. Discussion

In the Verification and Results section of this paper, we saw the message complexity being derived, both from a mathematical view using the pure definition of the algorithm(s), as well as through the implementation with counters and print statements. The values for the expected message amount, along with the realized ones, proved a perfect correlation, showing that our implementation adheres to the derived mathematical theory behind the algorithm.

Using a fitted quadratic tendency line on the regular Bully Algorithm, a perfect correlation between the mathematically modelled worst-case time complexity, presented previously, and the realized, can be deduced. The reason being that the R^2-value is 1, if rounded to two decimal places.

Our improved bully algorithm fits a linear tendency fit with the same R^2 value, namely 1 rounded to 2 decimals, also suggesting a perfect correlation between theory and practice. Looking at the absolute improvements of the 2 algorithms, it is clear that the modified Bully Algorithm introduces a significantly lower number of messages. The magnitude of this reduction is reflected in the relative improvement reaching nearly 90% after less than $P = 30$ on the network of interest, and converges towards a 100% as the amount of processes increases towards infinity, truly cementing the massive improvement it provides.

The regular Bully Algorithm is a great and simple implementation of a leader election method, with a major advantage in the fact that only processes with higher IDs than the current leader is properly involved in an election (more than 1 message), which lessens the total amount of messages sent. However, this is still the biggest issue with the regular Bully Algorithm; it requires a large amount of messages transmitted, with worst case complexity being $O(P^2)$, as previously shown.

This drawback is mitigated by the modified Bully Algorithm, which decreases the message complexity to $O(P)$, by choosing the biggest ID that responds to the election holder process.

The modified algorithm also ensures no parallel elections by adding a flag attribute to each process, essentially locking it for changes while an election is undergoing.

A drawback for the modified algorithm is the response time, how long the leader waits for a response before deeming a process dead. If we set the timer too high, the algorithm will be slow and inefficient, but setting it too low and we might miss delayed responses, e.g in a busy network. A possible redundancy in both versions is the election a process automatically starts, once it recovers or initializes. Perhaps this could be changed to be a query to any process, getting the PID of the current leader, then only beginning election if its own PID is higher than the one of the current leader.

There is limited information on the industry standard or state of the art leader election algorithms, but it is safe to say that consensus algorithms like the Raft consensus algorithm are among the top. The message complexity of Rafts leader election is also $O(P)$, since worst case is 1 process dead, resulting in $P-1$ election messages and $P-1$ responses, along with the victory messages $(P-1)$. A major advantage in the Raft election algorithm is that when a process is initialized it does not start an election, but instead just joins in and responds to messages.

Also, the leader does not have to wait for response from each process, as it does in either the Bully Algorithm, but instead just periodically pulses, allowing more leeway in case of busy networks. This leeway is also visible in elections, in which a process deems itself coordinator if it gets over half the votes, without needing to wait for any late responses.

The internal timer of each process also allows each process to effectively miss a response, and not be

deemed dead, given that it responds to any message before the timer runs out. This timer also ensures that really slow or overworked nodes will eventually be deemed dead, ideally lessening the load on them when reinitialized.

VI. Conclusion

The main objective of this report was to implement the Bully Algorithm as well as an improved Bully Algorithm, preferably in Python and verifying the superiority of the modified Bully Algorithm when considering the message complexity.

Using the definitions, we developed mathematical closed form notation for the message complexity of each of the two algorithms, and verified these using our implementation.

Conclusively, we showed the modified Bully Algorithm superiority to the regular Bully algorithm, using message complexity, linear vs quadratic.

The modified Bully Algorithm goes toe to toe with state of the art algorithms in terms of message complexity, but falls behind in the usability, especially in busy networks, as discussed previously. If the proposed changes were to be implemented to the modified algorithm however, it would drastically decrease the amount of elections (by around 50%), and possibly give the algorithm a place among the top Leader Election methods.

VII. References

[1] Abhinav Ajitsaria. What is the python global interpreter lock (gil)?, Oct 2020.

[2] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, January 1982.

[3] Philippe Kruchten. Architectural blueprints: The 4+1 view model of software architecture. *CoRR*, abs/2006.04975, 2020.

[4] Paulsingh Soundarabai, Ritesh Sahai, J. Thriveni, Venugopal K R, and Lalit Patnaik. Improved bully election algorithm for distributed systems. 02 2014.

[5] Kristoffer Plagborg Bak Sørensen. Kristoffer-pbs/distributed-system-project-1, Nov 2020.

[6] Wikipedia. Distributed computing — Wikipedia, the free encyclopedia, 2020. [Online; accessed 29-October-2020].