

**Formal Verification of Distributed Leader Election Algorithms with
Model Checking**

June 1, 2023

Student

Kevin Joshua Vinther
kevin20@student.sdu.dk

Supervisor

Marco Peressotti
Peressotti@imada.sdu.dk

Contents

1	Introduction	2
2	Overview	3
2.1	Terminology	3
2.2	Leader Election	3
2.3	Bully Algorithm	4
2.4	Ring Algorithm	5
2.5	TLA+	5
3	Bully Algorithm	6
3.1	Converting to TLA+	6
3.1.1	Deviations from pseudocode	6
3.2	Setup	6
3.3	Sending Messages	8
3.4	Receiving Messages	9
3.4.1	Handling Messages	10
3.5	Checking the leader condition	11
3.6	Next predicate and dying leader	11
3.6.1	Properties	12
4	Ring Algorithm	14
4.1	Thought Process	14
4.2	Setup	15
4.3	Helper functions	15
4.4	Event Handlers	16
4.5	HandleMessages	16
4.6	Checking The Leader Condition	17
5	Evaluation	18
6	Conclusion	19
A	Bully Algorithm Implementation	21
B	Ring Algorithm Implementation	22

Chapter 1

Introduction

TODO: Confidence 0/10:
🔔 What is the significance of Leader Election? What is the significance of TLA+? What do you solve?

Chapter 2

Overview

 TODO: Confidence 1/10


This is the main body section.

2.1 Terminology

 TODO: Confidence 1/10

Before explaining much of the implementation as well as how the algorithms works, it is important to show some of the terminology.

Failure models... Overlay... Logical... Classifications...

 is this necessary? maybe explaining when using words would be better...
probably

2.2 Leader Election

 TODO: Confidence 3/10


In several algorithms for distributed systems, a process may possess the role of the *leader*. A leader may be required in a system because algorithms typically are not completely symmetrical, and thus the leader can take the lead in initiating the algorithm. [1]

The aim of a leader election algorithm, thus, is to elect a leader from a set of processes. If this leader dies, it is the algorithms job to find a new leader.

This *report?* will search to implement two leader election algorithms: The *Bully Algorithm* and the *Ring Algorithm*¹

Both of the algorithms assume that:

- The network is reliable. Messages do not get lost.
- Nodes may fail at any time, including the leader.
- Fail-stop model. Failed nodes are removed from the system forever.

 maybesomething about how they are both on graphs?

2.3 Bully Algorithm

 TODO: Confidence 3/10

The bully algorithm is an algorithm which solves the problem of leader election. The bully algorithm assumes a complete undirected graph.[?] *Should I describe what a complete undirected graph is?*

 Should I be describing this?

The messages in the algorithm are:

- *election*: announce an election
- *alive*: response to *election*
- *victory*: sent by winner to announce victory

 Maybe this should be a figure? Easier to refer to

When a process P recovers from failure, or the failure detector indicates that the current leader has failed, P performs the following actions:

1. If P has the highest process ID, it sends a Victory message to all other processes and becomes the new leader. Otherwise, P broadcasts an Election message to all other processes with higher process IDs than itself.
2. If P receives no alive after sending an Election message, then it broadcasts a Victory message to all other processes and becomes the leader.

¹Also known as the Lelang, Chang and Roberts (LCR) algorithm, but commonly referred to as the Ring algorithm, as will be done in this report?

3. If P receives an alive from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)
4. If P receives an Election message from another process with a lower ID it sends an alive message back and if it has not already started an election, it starts the election process at the beginning, by sending an Election message to higher-numbered processes.
5. If P receives a victory message, it treats the sender as the leader.

2.4 Ring Algorithm

 TODO: Confidence 3/10

The Ring Algorithm is an algorithm that solves the Leader Election problem. In the bully algorithm, we assume a complete directed ring.

2.5 TLA+

 write something about distributed systems

TLA+ (Temporal Language of Actions Plus) is a language for modeling specifications, and widely used for proving algorithms. It has proven to be a very useful tool for proving algorithms, because of the way the language works, by forcing you to write mathematically correct implementations, that can be tested in every possible state.

Chapter 3

Bully Algorithm

3.1 Converting to TLA+

Since there is no specific pseudocode to the bully algorithm, but rather an instruction set for what process p should do when it wakes up, it has more so been a process of modelling from thoughts rather than converting from pseudocode.

3.1.1 Deviations from pseudocode

The second point in the algorithm described earlier states that, if process p receives no *alive* message after sending an Election message, then it broadcasts an *victory* message to all other processes. Since TLA⁺ does not allow for waiting a specified amount of time, I have made a major modification to the algorithm. Instead of waiting to see, it immediately checks if process P is the process with the highest ID alive, and broadcasts a *victory* message if this is the case. We allow ourself this because *fail-stop model?* assumes that all processes will know the failure of other processes, but it is not specified how. Therefore, we can assume that the process will know that it is the highest process. This is also done in the beginning of the algorithm, when a process checks if a leader is dead. However, the rest of the implementation is as true to the pseudocode *can I call it that?* as possible.

3.2 Setup



TODO: Confidence 6/10: Formatting, proof-reading, "we", articulation, maybe some of this is not necessary? too much explanation?

Before modeling the behavior of the specification, we set up the constants, variables and types *types? better explanation needed*. Looking at the specification for the algorithm, we need to model processes and messages.

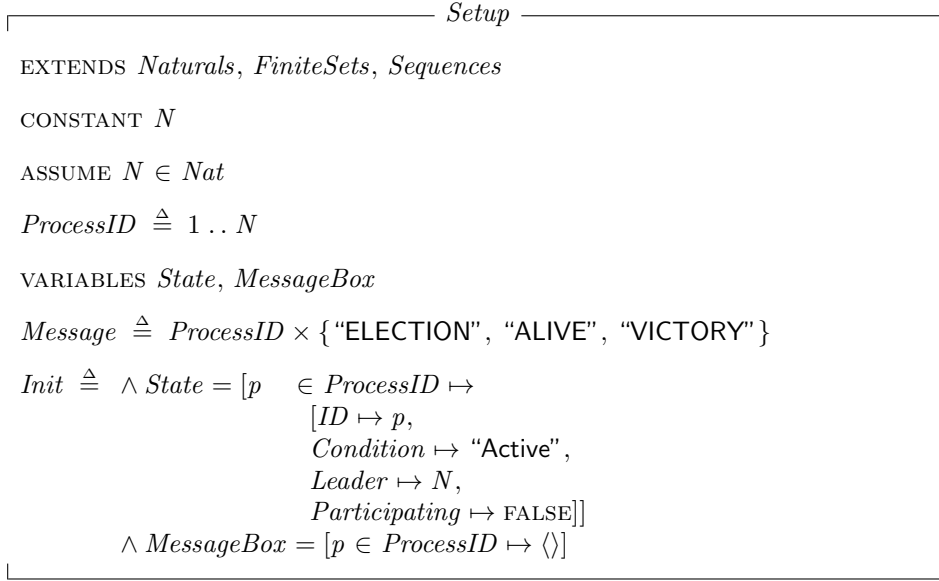


Figure 3.1: Setup for the Bully Algorithm

The algorithm assumes a complete undirected graph. We model this by creating a **State** variable, which contains a record of all *processes* and their fields. We define the **State** variable to be a sequence of records, each of which hold the following metadata in its fields:

- *ID*: The process has a unique ID. $ID \in ProcessID$, where $ProcessID$ is the total number of processes, defined as a range from 1 to the natural number N .
- *Condition*: A process is either dead or alive (called *active*). Due to the assumptions of the algorithm we assume that each process knows when another process is dead. $Condition \in \{ "Dead", "Active" \}$
- *Leader*: The process should know who it's leader is. The leader may be the process itself. $Leader \in ProcessID$
- *Participating*: The process is either participating in the election or not. This information is only used for the process itself. $Participating \in BOOLEAN$

Furthermore, we define a sequence, **MessageBox** which holds every processes received messages. The **MessageBox** maps from each process, p initially to an empty tuple, and later on to a sequence of tuples. These tuples are the **Messages**. We define a **Message** as $ProcessID \times \{ "ELECTION", "ALIVE", "VICTORY" \}$, thus an example of an "ELECTION" message from process 1 is: $\langle 1, "ELECTION" \rangle$

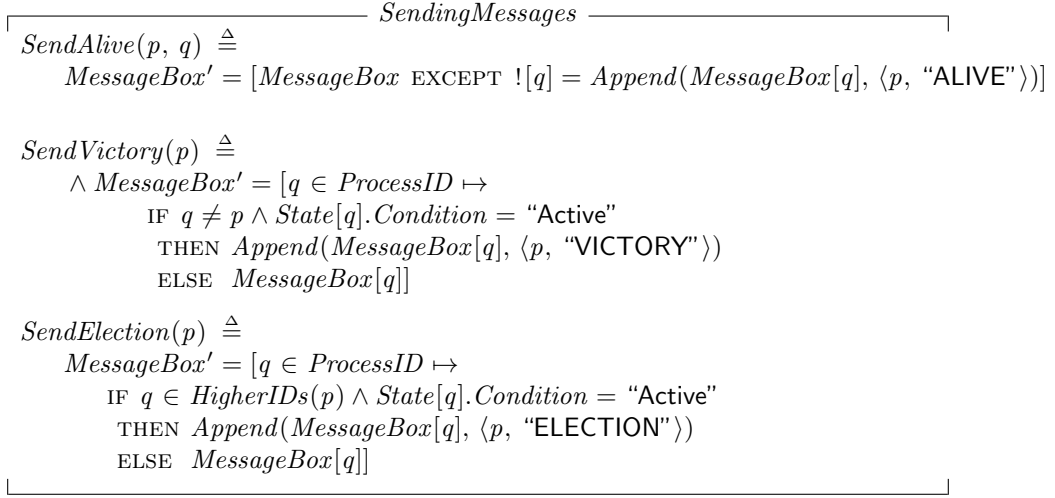


Figure 3.2: The *sender* functions

3.3 Sending Messages



How much of this should actually be described? Should this section maybe be after the explanation of handlers?

Along with the correct modeling of processes, the most important functionality in the algorithm is the sending, receiving and handling of messages.

We earlier defined the **MessageBox** to be a sequence of **Messages**. This makes the handling of messages extremely convenient. One of the core operations we make with messages are sending them. To send a message from process p to process q with the contents **ELECTION**, we simply append the tuple $\langle p, \text{"ELECTION"} \rangle$ to $MessageBox[q]$. In TLA⁺ this is done like so:

$MessageBox'$
 $= [MessageBox \text{ EXCEPT } ![q] = Append(\langle p, \text{"ELECTION"} \rangle)]$

This defines the $MessageBox$ variable to have $\langle p, \text{"ELECTION"} \rangle$ appended to $MessageBox[q]$ in the next state.



In earlier section, maybe write about why we have modelled the messages like this, rather than when using them? Why do we have sender e.g.

As discussed earlier, there are three different messages in the bully algorithm: *alive*, *victory*, and *election*.



Is the next section necessary or overflødig?

$$\text{HigherIDs}(p) \triangleq \{q \in 1 \dots \text{MaxAliveID} : q > p\}$$

Figure 3.3: Function which returns all IDs higher than p

The *alive* message is only sent in a case where a process receives an *election* message from a process with a lower id. Thus, no additional work is needed other than process q should receive an *alive* message from process p . This job is done by the *SendAlive* function.

The *victory* message is sent to all processes except for the sender. Therefore *SendVictory* sends a *victory* message to all processes except the sender. Note that it only sends to processes that are active. We can do this, because we assume a *fail-stop? i forgot* model, meaning that we know when processes die.

🔔 Why can't I just say $q \in (1..\text{MaxAliveID} \setminus p)$?

Similar to the *victory* message, the *election* message is sent to a select group of processes. This time it's all processes with a higher ID than p . This is achieved using a helper function, *HigherIDs*, which takes argument p and gives a list of all processes with a higher ID than p (see figure 3.3).

3.4 Receiving Messages

🔔 TODO: Confidence 5/10: Måske fungerer referencer til algoritme ikke helt?

$$\begin{aligned} \text{ReceiveAlive}(p) &\triangleq \wedge \text{State}[p].\text{Participating} = \text{TRUE} \\ &\quad \wedge \exists q \in \text{HigherIDs}(p) : \langle q, \text{"ALIVE"} \rangle \in \text{MessageBox}[p] \\ &\quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{FALSE}] \\ \\ \text{ReceiveElection}(p) &\triangleq \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\ &\quad \wedge (\exists q \in \text{LowerIDs}(p) : \langle q, \text{"ELECTION"} \rangle \in \text{MessageBox}[p] \\ &\quad \quad \wedge \text{State}[q].\text{Condition} = \text{"Active"} \\ &\quad \quad \wedge \text{SendAlive}(p, q)) \\ \\ &\quad \wedge \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Participating} = \text{TRUE}] \\ &\quad \wedge \text{SendElection}(p) \\ \\ \text{ReceiveVictory}(p, q) &\triangleq \text{State}' = [\text{State} \text{ EXCEPT } ![p].\text{Leader} = q] \end{aligned}$$

These functions take care of handling a message that has been received.

ReceiveAlive

If P receives an *Alive* from a process with a higher ID, it sends no further

messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)

This function makes sure that the process that has received the ALIVE message is currently participating in the election, and that the ALIVE message comes from a process with a higher ID. If both of these are true, the process is no longer participating in the election.

ReceiveElection

If P receives an Election message from another process with a lower ID it sends an ALIVE message back and if it has not already started an election, it starts the election process at the beginning, by sending an Election message to higher-numbered processes. TODO: Måske skal HigherID og LowerID bare forklares for sig selv?

This function first checks to make sure that the process is not already participating in the election. Then, it sees if the ELECTION message has come from a Lower ID, if this is the case it sends an “ALIVE” message to the process, if it is active. Again, making sure it is active is not a necessity, and only here to help with a clean MessageBox. TODO: Hvorfor virker den her funktion? Burde den ikke blive FALSE hvis det ikke er fra en election?

ReceiveVictory

If P receives a Victory message, it treats the sender as the coordinator.

ReceiveVictory changes the process' leader to be q .

3.4.1 Handling Messages

 TODO: Confidence 1/10: Dårlige forklaringer, dårlig grammatik etc.

$$\begin{aligned}
\text{HandleMessages}(p) \triangleq & \wedge \text{State}[p].\text{Condition} = \text{"Active"} \\
& \wedge \text{MessageBox}[p] \neq \langle \rangle \\
& \wedge \vee \wedge \text{State}[\text{Head}(\text{MessageBox}[p])[1]].\text{Condition} = \text{"Dead"} \\
& \quad \wedge \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![p] = \text{Tail}(\text{MessageBox}[p])] \\
& \quad \wedge \text{UNCHANGED State} \\
& \vee \wedge \text{Head}(\text{MessageBox}[p])[2] = \text{"VICTORY"} \\
& \quad \wedge \text{ReceiveVictory}(p, \text{Head}(\text{MessageBox}[p])[1]) \\
& \quad \wedge \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![p] = \text{Tail}(\text{MessageBox}[p])] \\
& \vee \wedge \text{Head}(\text{MessageBox}[p])[2] = \text{"ELECTION"} \\
& \quad \wedge \text{ReceiveElection}(p) \\
& \quad \wedge \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![p] = \text{Tail}(\text{MessageBox}[p])] \\
& \vee \wedge \text{Head}(\text{MessageBox}[p])[2] = \text{"ALIVE"} \\
& \quad \wedge \text{ReceiveAlive}(p) \\
& \quad \wedge \text{MessageBox}' = [\text{MessageBox} \text{ EXCEPT } ![p] = \text{Tail}(\text{MessageBox}[p])]
\end{aligned}$$

This function checks for new messages in a process' message box. It does this by looking at the first entry (the head) of the message box of process p . The first case is to clean up in case the process is dead. The only thing that is done, is to remove the message from the dead process. However, in the case where the process is not dead, the function looks at the contents of the message. It then calls the appropriate function depending the contents. After calling the

function it removes the latest message by modifying the messagebox to be the tail.

3.5 Checking the leader condition

TODO: Confidence 2/10: Dårlig grammatik, dårlig formulering etc.

$$\begin{aligned}
\text{CheckLeader}(p) \triangleq & \quad \wedge \text{State}[p].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"} \\
& \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\
& \wedge \text{IF } \text{ProcessHasTheHighestIDAlive}(p) = \text{TRUE} \\
& \quad \text{THEN } \text{SendVictory}(p) \\
& \quad \text{ELSE } \text{SendElection}(p) \wedge \text{State}' = [\text{State } \text{EXCEPT } ![p].\text{Participating} = \text{TRUE}]
\end{aligned}$$

This function checks whether or not the current leader is dead. It only does this, if there is the process checking is not participating in an election (meaning that we assume there is no election). We then use the helper function defined up in the Sends, to see if the process has the highest ID. If this is the case, we just send a victory message, otherwise we send an election message out, and change the process to be participating in the election. We can “just” say if the process has the highest ID, because we assume **TODO: hvad er det nu vi assumer siden vi bare kan sige det her? noget med at vi ved hvis processer er døde, vi ved bare ikke hvordan** This is also the reason this function is used, rather than not getting a response and then assuming the leader process is dead.

3.6 Next predicate and dying leader

 TODO: Confidence 2/10: For kort, dårligt beskrevet etc.

$$\begin{aligned}
\text{KillLeader}(p) \triangleq & \quad \wedge \text{State}[p].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Active"} \\
& \wedge \text{State}' = [\text{State } \text{EXCEPT } ![\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"}] \\
& \wedge \text{UNCHANGED } \text{MessageBox} \\
\\
\text{Next} \triangleq & \quad \exists p \in \text{ProcessID} : \\
& \quad \vee \text{CheckLeader}(p) \\
& \quad \vee \text{HandleMessages}(p) \\
& \quad \vee \text{KillLeader}(p)
\end{aligned}$$

The Next predicate is the predicate which chooses what should happen in the next state. Here we say that for some process p , we can either check if the leader is alive, handle any messages or kill the leader. Killing the leader is only a function to ensure the proper functioning of the spec.

$ \begin{aligned} TypeOK &\triangleq \forall p \in ProcessID : \\ &\wedge State[p].ID \in ProcessID \\ &\wedge State[p].Condition \in \{ "Active", "Dead" \} \\ &\wedge State[p].Leader \in ProcessID \\ &\wedge State[p].Participating \in BOOLEAN \\ &\wedge MessageBox \in [ProcessID \rightarrow Seq(Message)] \\ \\ OneLeader &\triangleq \forall p, q \in ProcessID : (State[p].Leader = p \\ &\quad \wedge State[q].Leader = q \\ &\quad \wedge State[p].Condition \neq "Dead" \\ &\quad \wedge State[q].Condition \neq "Dead") \\ &\quad \Rightarrow (p = q) \\ \\ UniqueID &\triangleq \forall p, q \in ProcessID : (State[p].ID = State[q].ID) \Rightarrow (p = q) \\ \\ Participating &\triangleq \forall p \in ProcessID : State[p].Participating = TRUE \Rightarrow State[p].Leader \neq p \end{aligned} $

How the invariants are set up:

The invariants are set up by depending on whether they are checking one process or two processes. If one process is checked, it checks for all processes p , while if they are checking two processes, they check for all processes p and q . Using this way, we make sure that each process gets treated correctly, and that we don't just select a random pair that might work at that moment.

Invariants will be checked to be true in every possible state.

Earlier in the report, we talked about TypeOK, as it is the invariant which makes sure that every variable holds legal values for this specification. It does this by checking each process in ProcessID (I.e., each process in general) and making sure that they are a part of the correct set. (TODO: FOR PROCESSID THIS IS REDUNDANT). This means that the Condition property should either be Active or Dead, the leader should be a legal process id, participating should be a boolean and the messagebox should be a mapping from processid to a sequence of messages.

The OneLeader invariant makes sure that if there exists two leaders, then those two leaders must be the same process. UniqueID makes sure that no process will have the same id.

Participating makes sure that if a process is participating, it is not the leader itself.

3.6.1 Properties

..Explain what properties are...

There are two kinds of temporal properties: safety properties, and liveness properties. Safety properties make sure that the system doesn't do bad things, and liveness makes sure our system always does a good thing.[?]

$$\begin{aligned}
& \text{ElectionTerminationImpliesSameLeader} \triangleq \forall p, q \in \text{ProcessID} : \\
& \quad (\text{State}[p].\text{Condition} = \text{"Active"} \\
& \quad \wedge \text{State}[q].\text{Condition} = \text{"Active"} \\
& \quad \wedge \text{State}[p].\text{Participating} = \text{FALSE} \\
& \quad \wedge \text{State}[q].\text{Participating} = \text{FALSE}) \\
& \quad \Rightarrow (\text{State}[p].\text{Leader} = \text{State}[q].\text{Leader}) \\
\\
& \text{MaxAliveID} \triangleq \text{CHOOSE } id \in \text{ProcessID} : \forall p \in \text{ProcessID} : \text{State}[p].\text{Condition} = \text{"Dead"} \vee id \geq \text{State}[p].ID \\
\\
& \text{HighestAliveProcessIsLeader} \triangleq \forall p \in \text{ProcessID} : \\
& \quad \text{State}[p].\text{Participating} = \text{FALSE} \Rightarrow \\
& \quad (\text{State}[\text{MaxAliveID}].\text{Condition} = \text{"Active"} \Rightarrow \text{State}[p].\text{Leader} = \text{MaxAliveID}) \\
\\
& \text{ElectionWillEnd} \triangleq \forall p \in \text{ProcessID} : \text{State}[p].\text{Participating} \leadsto (\Diamond(\text{State}[p].\text{Participating} = \text{FALSE})) \\
\\
& \text{EventuallyElection} \triangleq \forall p \in \text{ProcessID} : (\text{State}[p].\text{Condition} = \text{"Active"} \\
& \quad \wedge \text{State}[\text{State}[p].\text{Leader}].\text{Condition} = \text{"Dead"} \\
& \quad \wedge \text{State}[p].\text{Participating} = \text{FALSE}) \\
& \quad \leadsto \Diamond(\text{State}[p].\text{Participating} = \text{TRUE})
\end{aligned}$$

ElectionTerminationImpliesSameLeader is a safety property that ensures that when an election is done, everyone has the same leader.

HighestAliveProcessIsLeader is a safety property, tghat ensures that if a process is not participating in an election, then the highest alive process should be the leader.

ElectionWillEnd is a safety property that ensures that ensures that

Chapter 4

Ring Algorithm

...Also called LCR

4.1 Thought Process

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Similarities to Bully Algorithm

The Ring Algorithm and the Bully Algorithm are both algorithms for leader election, and thus many of the core functions and models can be reused. These include invariants, properties, processes, messages and more.

4.2 Setup

```

CONSTANT  $N$ 

ASSUME  $N \in \text{Nat}$ 

 $\text{ProcessID} \triangleq 1 \dots N$ 

VARIABLES  $\text{State}, \text{MBox}$ 

 $\text{Message} \triangleq \{ \text{"PROBE"}, \text{"SELECTED"} \} \times \text{ProcessID}$ 

 $\text{Init} \triangleq \wedge \text{State} = [p \in \text{ProcessID} \mapsto [$ 
     $\text{ID} \mapsto p,$ 
     $\text{Status} \mapsto \text{"Active"},$ 
     $\text{Leader} \mapsto N,$ 
     $\text{Participating} \mapsto \text{FALSE}]$ 
 $\wedge \text{MBox} = [p \in \text{ProcessID} \mapsto \langle \rangle]$ 

```

This setup is a lot like the setup of the Bully Algorithm, except the Message function is different. This is solely a syntactic change. In the Ring algorithm each message holds an “ID”, which is usually depicted as PROBE(ID) or SELECTED(ID). Thus, the change is made to reflect this, and also to avoid confusion with how the Bully Algorithm handles it, where the process is the sender of the message, not the id. The sender isn’t needed in the Ring Algorithm, as it is a ring structure and sends the message to the neighbour.

4.3 Helper functions

```

 $\text{HigherIDs}(p) \triangleq \{q \in \text{ProcessID} : q > p\}$ 
 $\text{LowerIDs}(p) \triangleq \{q \in \text{ProcessID} : q < p\}$ 

 $\text{SendMessage}(p, \text{id}, \text{msg}) \triangleq \text{MBox}' = [\text{MBox} \text{ EXCEPT } !p] = \text{Append}(\text{MBox}[p], \langle \text{msg}, \text{id} \rangle)$ 

 $\text{MaxAliveID} \triangleq \text{CHOOSE } \text{id} \in \text{ProcessID} : \forall p \in \text{ProcessID} : \text{State}[p].\text{Status} = \text{"Dead"} \vee \text{id} \geq \text{State}[p].\text{ID}$ 

 $\text{Neighbour}(p) \triangleq (p \% \text{MaxAliveID}) + 1$ 

```

$$2 + 2 = 4$$

$2 + 2 = 4$ The helper functions are quite alike the helper functions in the bully algorithm as well, with the most noticable differences being the SendMessage and Neighbour functions. The reason I decided for having a SendMessage function in the Ring Algorithm and not in Bully Algorithm, is that it aesthetically looks much prettier in the ring algorithm, and less cluttered, while it would only make it look more cluttered in the Bully Algorithm. The neighbour function is a helper function for finding the right-side neighbour of the process. Since the algorithm assumes a ring structure, the right-hand neighbour is just the process id plus one. However, since a leader might die, this is divided with the MaxAliveID, to make sure that if the right hand neighbour is dead, it is sent to

the lowest id instead. TODO: this should probably be changed, is in the real algorithm any process can die at any time.

4.4 Event Handlers

In the Ring algorithm there are two main events: receiving the Probe message and receiving the Selected message.

$$\begin{aligned}
ProcessProbe(p, id) &\triangleq \vee \wedge p < id \\
&\quad \wedge SendMessage(Neighbour(p), id, "PROBE") \\
&\vee \wedge p = id \\
&\quad \wedge State' = [State \text{ EXCEPT } ![p].Leader = id] \\
&\quad \wedge SendMessage(Neighbour(p), p, "SELECTED") \\
\\
ProcessSelected(p, id) &\triangleq \vee \wedge p \neq id \\
&\quad \wedge State' = [State \text{ EXCEPT } ![p].Leader = id, \\
&\quad \quad \quad ![p].Participating = FALSE] \\
&\vee \wedge p = id \\
&\quad \wedge State' = [State \text{ EXCEPT } ![p].Participating = FALSE]
\end{aligned}$$

ProcessProbe

According to the algorithm, in case a process receives a probe message, and the process id is lower than the id in the Probe message, the message is sent on the the next process. We do this using the `SnedMessage` and `Neighbour` helper functions. **TODO: THIS IS WHERE THE PROBLEM HAPPENS PROBABLY.** In the case of the ID being the same as the process ID, we assume that they are the new leader, and a `SELECTED` message with that process ID is sent, as well as their own leading being set to be themselves. If the process ID is greater than the ID in the message, we do nothing, as is specified by the algorithm.

ProcessSelected

ProcessSelected simply checks if the process ID is equal to the ID of the Selected message. If so, the process is declared the election is done. Otherwise, the message is sent on, to keep declaring the new leader.

4.5 HandleMessages

$$\begin{aligned}
HandleMessages(p) &\triangleq \wedge State[p].Status = \text{“Active”} \\
&\wedge MBox[p] \neq \langle \rangle \\
&\wedge \vee \wedge Head(MBox[p])[1] = \text{“PROBE”} \\
&\quad \wedge ProcessProbe(Neighbour(p), Head(MBox[p])[2]) \\
&\quad \wedge \text{IF } State[p].Participating = \text{FALSE THEN } State' = \\
&\quad \quad [State \text{ EXCEPT } ![p].Participating = \text{TRUE}] \text{ ELSE UNCHANGED } State \\
&\quad \wedge Print(\text{“We got here”}, \text{TRUE}) \\
&\quad \wedge MBox' = [MBox \text{ EXCEPT } ![p] = Tail(MBox[p])] \\
&\vee \wedge Head(MBox[p])[1] = \text{“SELECTED”} \\
&\quad \wedge ProcessSelected(p, Head(MBox[p])[2]) \\
&\quad \wedge MBox' = [MBox \text{ EXCEPT } ![p] = Tail(MBox[p])]
\end{aligned}$$

TODO: Change image

HandleMessage is mostly identical to the HandleMessage functionp from the

bully algorithm, except with changed messages, and in case of the message being a probe and the process' *participating* field being false, we change it to true.

4.6 Checking The Leader Condition

Chapter 5

Evaluation

Chapter 6

Conclusion

...While the Bully Algorithm worked fine, the Ring Algorithm did not.

Bibliography

- [1] Kshemkalyani, A., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2011), <https://books.google.dk/books?id=G7SZ32dPuLgC>

Appendix A

Bully Algorithm Implementation

Appendix B

Ring Algorithm Implementation