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

Chapter 1

Introduction

Today, more than ever before, distributed systems are crucial in computer infrastructure. However, they are also notoriously difficult to implement correctly. Even only a few interacting processes in a distributed system can lead to a tremendous amount of states. Catching each bug and edge case in the many possible states can take an enormous amount of time, and be exceedingly challenging.

Leader election is a distributed algorithm which requires that all processes in a distributed system agree on one specific process as their leader. This is crucial for many modern systems as distributed algorithms are typically not completely symmetrical, and therefore some process has to take the lead in initiating the algorithm, which a leader election algorithm can do. Leader election is also used because it saves resources by making sure that no process other than the leader replicates the algorithm initiation.

It is a challenge to verify that distributed algorithms work as intended in a distributed system using conventional tools today. Therefore, we use techniques from formal verification methods, because they help us test distributed algorithms in a way that no conventional tool can.

We seek to verify the correctness of two distributed leader election algorithms: the Bully Algorithm and the Ring Algorithm[1].

We do this using techniques from formal methods to check if there are any bugs, concurrency issues and security issues in the algorithm. Throughout this project the TLA$^+$ (Temporal Logic of Actions Plus) verification suite is used to model the Ring and Bully Algorithm

In the TLA$^+$ verification suite, the TLC model is included. The model checker verifies the algorithm by going through each possible state, and making sure that it produces satisfiable behavior, thus verifying if an algorithm works even in the cases that are close to impossible using conventional tools.

Thus, the goal of this project is to have modelled both the Bully Algorithm and the Ring Algorithm in the TLA$^+$ specification language, and have tested them using the TLC model checker.

[1]The Ring Algorithm is a familiar name for the Lelang, Chang and Roberts algorithm (also known as LCR). We will continue to use the name "Ring Algorithm".

Chapter 2

Overview

2.1 Leader Election

In several algorithms for distributed systems, a process may posses the role of the *leader*. Usually a leader may be required because distributed algorithms typically are not completely symmetrical. Thus, the leader can take the lead in initiating the algorithm. This also saves resources in the system, as it ensures that only the leader will the initiative in an algorithm, and that other processes won't do it as well. [**?**]

The aim of an algorithm for the Leader Election problem is to elect a leader from a set of processes in a way that all processes can agree on the same leader. It is crucial that all processes agree on the same leader, as if one process has the wrong leader, it can lead to many failures in the system.

This project focuses on the Bully Algorithm and the Ring Algorithm for Leader Election, both of which solve the problem. The Ring Algorithm works on a complete graph, with an undirected ring superimposed. The Bully Algorithm works on a complete undirected graph.

For the Bully and Ring Algorithm we 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.
- Messages are asynchronous.

2.2 Bully Algorithm

The Bully Algorithm solves the Leader Election problem using a complete, undirected graph as its overlay. The elected leader will either be the leader with the lowest ID, or the highest ID. In this report, we go with the assumption that it is the highest ID.

In the algorithm, there exists three different types of message to be sent and received. These messages initiate behavior on the receiving process, depending on what the contents of the message is. The messages are:

- *election*: sent to announce an election
- *alive*: sent to respond to an *election* message

- *victory*: sent by winner to announce victory

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 *id*s 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.
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.

In this version of the pseudocode, we assume that the leader will eventually become the process with the highest id alive.

2.3 Ring Algorithm

The Ring Algorithm, like the Bully Algorithm, solves the problem of leader election. However, unlike the Bully Algorithm, the Ring Algorithm assumes a superimposed undirected ring. This means that we assume the processes to be in a ring, and create the implementation from this assumption.

In the Ring Algorithm, there exists two different types of messages. Both messages dictate the behavior of the receiver:

- *probe(id)*: search for the leader
- *selected(id)*: announce the result

You will notice in the messages, that, unlike in the Bully Algorithm, they have attached an *id*. While it may read as if it is a function, it is just an attachment to the message. The *id* is used to identify who originally sent the message.

Furthermore, the Ring Algorithm requires a *boolean* variable, called *participate*. The variable indicates whether or not process P_i participates in the election.
When process P_i wakes up to participate in the election:

1. Send *probe(i)* to right neighbour
2. *participate* $\leftarrow$ True

When *probe(k)* message arrives from the left neighbour P_j:

1. If *participate* = False then *participate* $\leftarrow$ True.
2. if $i > k$ then discard the probe

3. else if $i < k$ then forward *probe(k)* to right neighbor

4. else if $i = k$ then declare i is the leader; circulate *selected(i)* to right neighbour;

When a *selected(x)* message arrives from left neighbour:

1. if $x \neq i$ then note x as the leader and forward message to right neighbor

2. else do not forward the *selected* message.

([**?**])

In this algorithm, like in the Bully Algorithm, we assume that the leader will be the process with the highest id alive.

Chapter 3

Bully Algorithm

3.1 Converting to TLA+

Since we have based ourself on an instruction set rather than pseudocode for what process p should do, it has been more a process of modelling the algorithm correctly, rather than converting it 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 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 the *fail-stop* 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 set of instructions as possible. Furthermore, I have made a modification to point 4. However, since this is not done because of limitations of TLA$^+$, it is discussed in Section 3.4.

3.2 Setup

Before modeling the behavior of the specification, we set up the constants, variables and initialize them in the *Init* function. Looking at the specification for the algorithm, we need to model processes and messages.

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. $Leader \in ProcessID$

Setup

EXTENDS *Naturals*, *FiniteSets*, *Sequences*

CONSTANT N

ASSUME $N \in Nat$

$ProcessID \triangleq 1 \,..\, N$

VARIABLES *State*, *MessageBox*

$Message \triangleq ProcessID \times \{\text{“ELECTION”}, \text{“ALIVE”}, \text{“VICTORY”}\}$

$$
\begin{aligned}
Init \triangleq\ & \wedge State = [p \in ProcessID \mapsto \\
& \qquad [ID \mapsto p, \\
& \qquad Condition \mapsto \text{“Active”}, \\
& \qquad Leader \mapsto N, \\
& \qquad Participating \mapsto \text{FALSE}]] \\
& \wedge MessageBox = [p \in ProcessID \mapsto \langle\rangle]
\end{aligned}
$$

Figure 3.1: Setup for the Bully Algorithm

- *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 **Message**s. We define a **Message** as ProcessID $\times$ {“ELECTION”, “ALIVE”, “VICTORY”}, thus an example of an “ELECTION” message from process 1 is: $\langle 1,$ “ELECTION” $\rangle$.

3.3 Sending Messages

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 **Message**s. 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 $\langle p,$ “ELECTION” $\rangle$ to *MessageBox[q]*. In TLA$^+$ this is done like so:

```
MessageBox'
    = [MessageBox EXCEPT ![q] = Append(<<p, ''ELECTION''>>)]
```

This defines the MessageBox variable to have $\langle p,$ “ELECTION” $\rangle$ appended to *MessageBox[q]* in the next state.

In Figure 3.2 you will notice two different types of sending predicates: one which is simply “SendX”, and one which is “SendXAndTail”. The AndTail variant does the same as the other, except it also removes the newest message in process p's inbox. This is done as to not clutter the inbox with irrelevant messages.

SendingMessages

$SendAliveAndTail(p,\ q) \triangleq$
$MessageBox' = [MessageBox \text{ EXCEPT } ![q] = Append(MessageBox[q], \langle p, \text{“ALIVE”}\rangle),$
$![p] = Tail(MessageBox[p])]$

$SendVictory(p) \triangleq$
$MessageBox' = [q \in ProcessID \mapsto$
IF $q \neq p \land State[q].Condition = \text{“Active”}$
THEN $Append(MessageBox[q], \langle p, \text{“VICTORY”}\rangle)$
ELSE $MessageBox[q]]$

$SendVictoryAndTail(p) \triangleq$
$MessageBox' = [q \in ProcessID \mapsto$
IF $State[q].Condition = \text{“Active”}$
THEN IF $q = p$ THEN $Tail(MessageBox[q])$ ELSE $Append(MessageBox[q], \langle p, \text{“VICTORY”}\rangle)$
ELSE $MessageBox[q]]$

$SendElection(p) \triangleq$
$MessageBox' = [q \in ProcessID \mapsto$
IF $q \in HigherIDs(p) \land State[q].Condition = \text{“Active”}$
THEN $Append(MessageBox[q], \langle p, \text{“ELECTION”}\rangle)$
ELSE $MessageBox[q]]$

Figure 3.2: The *sender* functions

MaxAliveID

$MaxAliveID \triangleq \text{CHOOSE } id \in ProcessID :$
$\forall p \in ProcessID : State[p].Condition = \text{“Dead”}$
$\lor\ id \geq State[p].ID$

Figure 3.3: Function that returns the highest ID alive

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

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 *SendAliveAndTail* function. Since it as an “AndTail” function it also removes the newest message by setting the message box of p to be the tail of the message box.

The *SendVictory* function is similar, only it sends the *victory* message to all processes except the sender. We assume a fail-stop model, meaning that the other processes learn when a process dies, allowing us to send only to the alive processes. This function also has a “AndTail” variant, which is identical except it tails p's message box.

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* (see figure 3.3.)

HigherIDs

$$HigherIDs(p) \triangleq \{q \in 1 \,..\, MaxAliveID : q > p\}$$

Figure 3.4: Function which returns all IDs higher than p

HandleReceivedMessages

$$\begin{aligned}
ReceiveAlive(p,\, q) \triangleq\ & \land State[p].Participating = \text{TRUE} \quad \text{Make sure they are already participating} \\
& \land p > q \\
& \land State' = [State \text{ EXCEPT } ![p].Participating = \text{FALSE}] \quad \text{No longer participate}
\end{aligned}$$

$$\begin{aligned}
ReceiveElection(p,\, q) \triangleq\ & \lor \land p = MaxAliveID \\
& \quad \land SendVictoryAndTail(p) \\
& \quad \land State' = [State \text{ EXCEPT } ![p].Participating = \text{FALSE}, \\
& \qquad\qquad\qquad\qquad\quad ![p].Leader = p] \\
& \lor \land p \neq MaxAliveID \\
& \quad \land \lor \land State[p].Participating = \text{TRUE} \\
& \qquad\ \land SendAliveAndTail(p,\, q) \\
& \qquad\ \land \text{UNCHANGED } State \\
& \quad\ \lor \land State[p].Participating = \text{FALSE} \\
& \qquad\ \land SendAliveAndTail(p,\, q) \\
& \qquad\ \land State' = [State \text{ EXCEPT } ![p].Participating = \text{TRUE}]
\end{aligned}$$

$$ReceiveVictory(p,\, q) \triangleq State' = [State \text{ EXCEPT } ![p].Leader = q] \quad \text{Set the leader of receiver to } p \text{ sender}$$

Figure 3.5: Message Handlers

3.4 Handling Received Messages

When a message is received, a process will periodically check the *HandleMessages* function, which will then execute the correct functions depending on the message. The functions in Figure 3.5 describe the behavior when receiving a specific message.

Points 3, 4, and 5 rom the pseudocode describe what to do when receiving messages:

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.

The function *ReceiveAlive* addresses point 3. It checks if the message is from a higher id than itself, and if so, it modifies its participation status to be FALSE.

The function *ReceiveElection* addresses point 4. However, you may notice that it does not correctly follow point 4. Point 4 explicitly states that **"...if it has not already started an election, it starts the elction process at the beginning, by sending an Election message to higher-numbered processes"**. Instead of doing this, it only sends an alive

DelegateMessages

$$
\begin{array}{l}
HandleMessages(p) \triangleq \text{LET} \\
\quad sender \triangleq Head(MessageBox[p])[1] \\
\quad msg \triangleq Head(MessageBox[p])[2] \\
\quad \text{IN} \\
\quad\quad \wedge\ State[p].Condition = \text{“Active”} \\
\quad\quad \wedge\ MessageBox[p] \neq \langle\rangle \quad \text{Message box shouldn't be empty} \\
\quad\quad \wedge\ \vee\ \wedge\ State[sender].Condition = \text{“Dead”} \\
\quad\quad\quad\quad\ \wedge\ MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])] \quad \text{Remove first message} \\
\quad\quad\quad\quad\ \wedge\ \text{UNCHANGED } State \\
\quad\quad\quad\ \vee\ \wedge\ msg = \text{“VICTORY”} \quad \text{If the message is } VICTORY \\
\quad\quad\quad\quad\ \wedge\ ReceiveVictory(p,\ sender) \\
\quad\quad\quad\quad\ \wedge\ MessageBox' = [MessageBox \text{ EXCEPT } ![p] = \langle\rangle] \quad \text{Remove all messages} \\
\quad\quad\quad\ \vee\ \wedge\ msg = \text{“ELECTION”} \quad \text{If the message is } ELECTION \\
\quad\quad\quad\quad\ \wedge\ ReceiveElection(p,\ sender) \quad \text{Tail is done incide } ReceiveElection \\
\quad\quad\quad\ \vee\ \wedge\ msg = \text{“ALIVE”} \quad \text{If the message is } ALIVE \\
\quad\quad\quad\quad\ \wedge\ ReceiveAlive(p,\ sender) \quad \text{Handle in } ReceiveAlive \\
\quad\quad\quad\quad\ \wedge\ MessageBox' = [MessageBox \text{ EXCEPT } ![p] = Tail(MessageBox[p])] \quad \text{Remove first message}
\end{array}
$$

Figure 3.6: Delegating Functions

message, and, if its own participation status is FALSE, it sets it to be TRUE instead. This change is done because it is a redundancy in the algorithm, and will only lead to a number of irrelevant *election* messages. Furthermore, we have deviated from point 4 in the sense that we check if the process has the highest ID alive, and if so, it declares victory. This is done because we assume a fail-stop model.

The *ReceiveVictory* function (Figure 3.5) addresses point 5 by setting the leader to the sender of the *victory* message.

3.5 Delegating Functions with HandleMessages

In the *Next* predicate of the specification, one of the actions a process can take is to check its messages. This is done via the *HandleMessages* function. If process p has any messages in their inbox, one of 4 things can happen, depending on the type of message or the condition of the sender.

1. If the sender is dead, we ignore the message and delete it from the inbox of process p.

2. If the content of the message is “VICTORY”, we delegate work to the *ReceiveVictory* function and remove all messages from process p because we do not care about other message at this point.

3. If the content of the message is “ELECTION”, we delegate the rest of the work to the *Receive-Election* function.

4. If the content of the message is “ALIVE”, we delegate work to the *ReceiveAlive* function and delete the message from the inbox of process p.

$$
\begin{array}{l}
CheckLeader(p) \triangleq \quad \land\ State[p].Condition = \text{“Active”} \\
\qquad\qquad\qquad\quad \land\ State[State[p].Leader].Condition = \text{“Dead”} \\
\qquad\qquad\qquad\quad \land\ \lor\ \land\ p = MaxAliveID \\
\qquad\qquad\qquad\qquad\qquad\ \land\ SendVictory(p) \\
\qquad\qquad\qquad\qquad\qquad\ \land\ State' = [State\ \text{EXCEPT}\ ![p].Participating = \text{FALSE},\ ![p].Leader = p] \\
\qquad\qquad\qquad\qquad\ \lor\ \land\ p \neq MaxAliveID \\
\qquad\qquad\qquad\qquad\qquad\ \land\ State[p].Participating = \text{FALSE} \\
\qquad\qquad\qquad\qquad\qquad\ \land\ SendElection(p) \\
\qquad\qquad\qquad\qquad\qquad\ \land\ State' = [State\ \text{EXCEPT}\ ![p].Participating = \text{TRUE}]
\end{array}
$$

Figure 3.7: Checking the condition of the leader

NextPredicateAndKillLeader

$$
\begin{array}{l}
KillLeader \triangleq \\
\quad \land\ State[MaxAliveID].Leader = MaxAliveID \\
\quad \land\ State' = [State\ \text{EXCEPT}\ ![MaxAliveID].Condition = \text{“Dead”},\ ![MaxAliveID].Participating = \text{FALSE}] \\
\quad \land\ Cardinality(\{p \in ProcessID : State[p].Condition = \text{“Active”}\}) \geq 2 \\
\quad \land\ \text{UNCHANGED}\ MessageBox \\
\\
Next \triangleq \quad \lor\ KillLeader \\
\qquad\qquad \lor\ \exists\, p \in 1\,..\,MaxAliveID : CheckLeader(p) \lor HandleMessages(p)
\end{array}
$$

Figure 3.8: Next predicate and KillLeader

3.6 Checking the leader condition

One of the actions a process can take in the *next* state is to check the condition of the leader. In the pseudocode, it is specified that “When a process P recovers from failure, or the failure detector indicates that the current leader has failed...”. Since we assume that the processes do not recover from failure, and know that a process is dead, we use the *CheckLeader* (Figure 3.7) function instead.

The function only runs if the process checking is active, and the leader is dead. If the process is active, the function checks whether process p is the process with the highest id. If p has the highest id, it sends a *victory* message, and declares itself the new leader. If p is not the process with the highest id, we first make sure that it is not currently participating in any election, as to not accidentally start multiple elections in one cycle. Then process p sends an election message, and sets its own Participation status to True, as it is now in an election.

3.7 Next predicate and killing the leader

The Next action (Figure 3.8) is the action which chooses what should happen in the next state. For this specification there are two options. Either, the leader is killed, or a process handles its messages or checks its leaders status.

In case the choice is made to kill the leader, we make sure that the leader sees itself as the leader, and if they do we change its condition in the next state to be dead and its participation status to be False, effectively killing it. This is only done if there are 2 or more processes alive.

In case the decision to kill the leader is not chosen, a process is chosen to either check it's leader or handle it's messages.

3.8 Invariants and properties

Invariants
In TLA$^+$, an *invariant* is a statement that must be true on every state of the program[**?**]. One common type of invariant is the *TypeOK* invariant (Figure 3.9). *TypeOK* makes sure that the types never go out of a specified domain or change to an unwanted value. In the case of this specification this is true for the *State* and *MessageBox* variables. For the *State* variable, it will hold true that:

- A process' *id* is a part of the *ProcessID*[1] set
- A process' condition is either *Active* or *Dead*
- A process' *leader* is a part of the *ProcessID* set
- A process' participation status is a BOOLEAN

For the message box it holds that each ID will map to a sequence of messages[2].

In this specification, we also have an invariant called *UniqueID* which makes sure that no two processes hold the same ID. Furthermore, we also have an invariant called *Participating* which states that, if a process is participating in an election, it cannot be the leader itself, or see itself as leader.
Properties
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.[**?**]

We have three properties: two safety properties and one liveness property. The safety properties are *ElectionTerminationImpliesSameLeader*, and *HighestAliveProcessIsLeader*. *ElectionTerminationImpliesSameLeader* ensures that when two processes that have been in an election eventually get out of the election, they will have the same leader. *HighestAliveProcessIsLeader* ensures that if a process is not participating in an election, it's leader is the process with the highest id alive.

The liveness property, *ElectionWillEnd* ensures that an election will eventually end.

3.9 Temporal Formula

To run the specification with the model checker, the temporal formula in Figure 3.10 is used. The reason a temporal formula is used rather than just specifying the Initial predicate and Next-state relation is to use *strong fairness*. Without strong fairness, the model checker might just decide to stop the entire system. That is an intended feature, but it is not what we want. Strong fairness says that if an action is always enabled, it eventually happens.

[1] See figure 3.1 for the definition of ProcessID
[2] See figure 3.1 for the defintion of Message

InvariantsandProperties

Invariants

Each variable should be within the type constraints

$$\begin{aligned} TypeOK \triangleq\ & \forall\, p \in ProcessID : \\ & \land\ State[p].ID \in ProcessID \\ & \land\ State[p].Condition \in \{\text{“Active”},\ \text{“Dead”}\} \\ & \land\ State[p].Leader \in ProcessID \\ & \land\ State[p].Participating \in \text{BOOLEAN} \\ & \land\ MessageBox \in [ProcessID \to Seq(Message)] \end{aligned}$$

Every process should have a unique *ID*

$$UniqueID \triangleq \forall\, p,\ q \in ProcessID : (State[p].ID = State[q].ID) \Rightarrow (p = q)$$

If a process is participating in the election, then it should not be the leader

$$Participating \triangleq \forall\, p \in ProcessID : State[p].Participating = \text{TRUE} \Rightarrow State[p].Leader \neq p$$

Properties

$$\begin{aligned} ElectionTerminationImpliesSameLeader \triangleq\ & \forall\, p,\ q \in ProcessID : \\ & (State[p].Condition = \text{“Active”} \\ & \land\ State[q].Condition = \text{“Active”} \\ & \land\ State[p].Participating = \text{FALSE} \\ & \land\ State[q].Participating = \text{FALSE}) \\ & \Rightarrow (State[p].Leader = State[q].Leader) \end{aligned}$$

The highest alive process should become the leader

$$\begin{aligned} HighestAliveProcessIsLeader \triangleq\ & \forall\, p \in ProcessID : \\ & State[p].Participating = \text{FALSE} \Rightarrow \\ & (State[MaxAliveID].Condition = \text{“Active”} \Rightarrow State[p].Leader = MaxAliveID) \end{aligned}$$

Eventually, the election will end

$$ElectionWillEnd \triangleq \forall\, p \in ProcessID : State[p].Participating \rightsquigarrow (\Diamond(State[p].Participating = \text{FALSE}))$$

Figure 3.9: Invariants and Properties in the Bully Algorithm

TemporalFormula

$$Spec \triangleq Init \land \Box[Next]_{\langle State,\, MessageBox\rangle} \land \mathrm{SF}_{\langle State,\, MessageBox\rangle}(Next)$$

Figure 3.10: Temporal Formula

Chapter 4

Ring Algorithm

4.1 Similarities to Bully Algorithm

The Ring Algorithm and the Bully Algorithm are both similar in many ways. One similarity which carries over from our implementation of the Bully Algorithm to the Ring Algorithm is the modeling of the graphs. Since the Ring Algorithm is also an algorithm on graphs, we can reuse a lot of the setup. Furthermore, the Ring Algorithms superimposed overlay is an undirected ring. We will not need to model it is a ring, even though it is a ring. Instead, we assume it to be a ring, and write the specification as if it was a ring.

4.2 Setup

The setup (See Figure 4.1) for the Ring Algorithm is almost identical to that of the Bully Algorithm. The only difference is how we define the **Message**s. A message in the Ring Algorithm has been changed in two ways. First, we don't send the same messages in the Ring Algorihtm. The messages in the Ring Algorithm are *probe* and *selected*. Second, we switch the order of message and process. In the Bully Algorithm it made sense to have the ProcessID as the first part of the message, since it is the sender of the message. In the Ring Algorithm, the process id is the ID of the message, and not the sender. Since both messages in the Ring Algorithm have an ID attached to them, the messages instead have the process on the right side. However, this is purely a syntactic choice.

4.3 Neighbours and Sending Messages

Neighbour
In a ring topology, each process has a left-side neighbour and a right-side neighbour. For the Ring Algorithm we only care about the right-side neighbour. Thus, the *Neighbour* function (Figure 4.2 returns the left-side neighbour. However, there is a flaw in this function, that makes it deviate from the original pseudocode. It assumes that only the leaders die, and no non-leaders will die. This is the only deviation from the original specification.

Sending Messages
We deviate a lot from the Bully algorithm in how messages are sent. Instead of having functions for specific messages, we have two functions *SendMessage* and *SendMessageAndTailMessageBox*.

Setup

```
EXTENDS Naturals, FiniteSets, Sequences, TLC

CONSTANT N

ASSUME N ∈ Nat

ProcessID ≜ 1 .. N

VARIABLES State, MessageBox

Message ≜ {"PROBE", "SELECTED"} × ProcessID

 Initializes the variables: State and MessageBox
Init ≜ ∧ State = [p ∈ ProcessID ↦ [
                    ID ↦ p,
                    Condition ↦ "Active",
                    Leader ↦ N,
                    Participating ↦ FALSE]]
       ∧ MessageBox = [p ∈ ProcessID ↦ ⟨⟩]
```

Figure 4.1: The Setup for the Ring Algorithm

Neighbour

```
Neighbour(p) ≜
    LET AliveProcesses ≜ {q ∈ ProcessID : State[q].Condition = "Active"}
    IN
    IF p = MaxAliveID THEN CHOOSE q ∈ AliveProcesses : TRUE ELSE
            CHOOSE neighbour ∈ AliveProcesses :
                    ∧ p < neighbour
                    ∧ ∀ q ∈ AliveProcesses : q ≤ p ∨ neighbour ≤ q
```

Figure 4.2: Neighbour function

SendingMessages

```
SendMessageAndTailMessageBox(p, q, id, msg) ≜
        MessageBox' = [MessageBox EXCEPT ![q] = Append(MessageBox[q], ⟨msg, id⟩),
                                         ![p] = Tail(MessageBox[p])]

 Simple SendMessage. Sends message msg to q with id id.
SendMessage(q, id, msg) ≜
        MessageBox' = [MessageBox EXCEPT ![q] = Append(MessageBox[q], ⟨msg, id⟩)]

SendMessageAndEmptyMessageBox(p, q, id, msg) ≜
        MessageBox' = [MessageBox EXCEPT ![q] = Append(MessageBox[q], ⟨msg, id⟩), ![p] = ⟨⟩]
```

Figure 4.3: Sending Messages

HandleMessages

```
HandleMessages(p) ≜ LET
        id ≜ Head(MessageBox[p])[2]
        msg ≜ Head(MessageBox[p])[1]
        IN
        ∧ State[p].Condition = "Active"  Only if the process is active
        ∧ MessageBox[p] ≠ ⟨⟩             The message box shouldn't be empty
        ∧ ∨ ∧ State[id].Condition = "Dead"
            ∧ MessageBox' = [MessageBox EXCEPT ![p] = Tail(MessageBox[p])]
            ∧ UNCHANGED State
          ∨ ∧ msg = "PROBE"  If the message is PROBE
            ∧ ∨ ∧ State[p].Participating = FALSE  Make sure the participation status is TRUE
                ∧ State' = [State EXCEPT ![p].Participating = TRUE]
              ∨ ∧ State[p].Participating = TRUE
                ∧ UNCHANGED State
            ∧ ∨ ∧ p = id  If p = id, send SELECTED
                ∧ SendMessageAndEmptyMessageBox(p, Neighbour(p), p, "SELECTED")
              ∨ ∧ p > id  ∧ State[p].Participating = FALSE  If p > id ignore message
                ∧ SendMessageAndTailMessageBox(p, Neighbour(p), p, "PROBE")
              ∨ ∧ p > id ∧ State[p].Participating = TRUE
                ∧ MessageBox' = [MessageBox EXCEPT ![p] = Tail(MessageBox[p])]
              ∨ ∧ p < id  If p < id send probe to neighbour
                ∧ SendMessageAndTailMessageBox(p, Neighbour(p), id, "PROBE")
          ∨ ∧ msg = "SELECTED"  If the message is SELECTED
            ∧ ∨ ∧ p ≠ id  If p ≠ id, then send SELECTED to neighbour
                ∧ State' = [State EXCEPT ![p].Leader = id, ![p].Participating = FALSE]
                ∧ SendMessageAndEmptyMessageBox(p, Neighbour(p), id, "SELECTED")
              ∨ ∧ p = id  if p = id then declare itself the leader
                ∧ State' = [State EXCEPT ![p].Leader = id, ![p].Participating = FALSE]
                ∧ MessageBox' = [MessageBox EXCEPT ![p] = ⟨⟩]
```

Figure 4.4: HandleMessages

In the Bully Algorithm, a lot had to be done when sending each message. In the Ring Algorithm, messages are only sent to neighbours, thus requiring less work choosing who to send to. In the Bully Algorithm this work was done in the sender functions, but this work is not needed in the Ring Algorithm.

The *SendMessage* function updates the MessageBox of a process with a Message. The *SendMessageAndTailMessageBox* is similar, except it also removes the first message in the MessageBox of p. This is done, because the *HandleMessages* function requries both sending messages and deleting in the same state. However, in TLA$^+$ you cannot update a variable twice in the same state, thus requiring a function like *SendMessageAndTailMessageBox* to update both message boxes at the same time. *SendMessageAndEmptyMessageBox* is similar to *SendMessageAndTailMessageBox*, but instead of deleting just the first message, it deletes all messages.

4.4 HandleMessages

Instead of having separate functions for handling the different messages, all of this is done in the *HandleMessages* function (Figure 4.4).

The function checks if there are any messages for process p, and then proceeds to check what kind of message the first message is.

Receiving a *probe* message
In case the message received is *probe*, we check first if process p is participating. If it is not, we change the *participating* field to be true.

Afterwards, we compare p with the *id* from the message. If $p = id$, then the probe has circled around all processes, and we send the "SELECTED" message to the neighbour. The message box of p is then emptied. We empty the message box to ensure that there aren't any excess messages that will harm the election.
If $p > id$, we deviate from the original pseudocode. The original pseudocode states that: "if p ¿ id, discard the probe." However, this leads to cases with infinite loops of *probe* messages. Thus, instead what we do is, if $p > id$ and p is not participating in the election, we send a new *probe* message with p as the id, so that it can become the leader. If p however is participating, we just discard the message.
If $p < id$ we send the probe to the right neighbour, and delete the message.

Receiving a *selected* message
In case the msg is "SELECTED" we do not need to check whether or not the process is participating in the election, as they will have participated no matter what, as the *probe* message has been all around the ring. If the $id \neq p$ then we note *id* to be the leader and set our participation status to be FALSE. We also send the *selected* message to our right neighbour, so that everyone knows who the new leader is. However, if $p = id$ then p is the leader, and he is noted as such by himself. There is a slight deviation from the pseudocode here. In the pseudocode it is stated that p is noted by itself to be the leader, when it first sends the *selected* message to its right neighbour. However, due to the limitations of TLA$^+$, I have decided against doing it there, as it would require a check of participation in each equality check with the *id* when it receives the *probe* message. This solution is simply easier to look at. The desired outcome will be the same, i.e. every process having the same leader and not being participating.

4.5 Checking The Leader Condition

When we check if the leader is dead in the Ring Algorithm (See Figure 4.5), we do it a bit differently than in the Bully Algorithm. Here, the only check we, other than to see if the leader is dead, do is to see if there is only one process remaining. If this is the case, we set that one process as the leader, and its participation status to be FALSE ending the election (once and for all). However, if this is not the case, we send a *probe* message to the right neighbour and set ps participation status to TRUE.

4.6 Next predicate and killing the leader

KillLeader and the next predicate are identical to the Bully Algorithm.

CheckLeader

$$
\begin{array}{l}
CheckLeader(p) \triangleq \\
\quad \land State[p].Condition = \text{“Active”} \\
\quad \land State[State[p].Leader].Condition = \text{“Dead”} \\
\quad \land \lor \land MaxAliveID = 1 \\
\qquad\quad \land State' = [State \text{ EXCEPT } ![p].Leader = p,\ ![p].Participating = \text{FALSE}] \\
\qquad\quad \land \text{UNCHANGED } MessageBox \\
\qquad \lor \land MaxAliveID \neq 1 \\
\qquad\quad \land SendMessage(Neighbour(p),\ p,\ \text{“PROBE”}) \\
\qquad\quad \land State' = [State \text{ EXCEPT } ![p].Participating = \text{TRUE}]
\end{array}
$$

Figure 4.5: CheckLeader

NextPredicateandKillLeader

$$
\begin{array}{l}
KillLeader \triangleq \\
\quad \land State[MaxAliveID].Leader = MaxAliveID \\
\quad \land State' = [State \text{ EXCEPT } ![MaxAliveID].Condition = \text{“Dead”}] \\
\quad \land Cardinality(\{p \in ProcessID : State[p].Condition = \text{“Active”}\}) \geq 2 \\
\quad \land \text{UNCHANGED } MessageBox \\
\\
Next \triangleq \lor KillLeader \\
\qquad\quad \lor \exists\, p \in 1 \,..\, MaxAliveID : HandleMessages(p) \lor CheckLeader(p)
\end{array}
$$

Figure 4.6: Next predicate and KillLeader

4.7 Invariants and Properties

The Ring Algorithm does not introduce any new invariants or properties (Figure 4.7) that the Bully Algorithm does not already have. However, the invariant *Participating* (Figure 3.9) is not present. The Participating invariant does not work in the Ring Algorithm, since it checks whether or not a leader is participating, however, a leader will participate in the election if it receives a *probe* message. However, all of the other invariants and properties are identical to those of the Bully Algorithm.

4.8 Temporal Formula

Like in the Bully Algorithm, the Ring Algorithm also uses a temporal formula rather than stating the initial predicate and the next-state relation in the model checker. Like in the Bully Algorithm, this is done to use *strong fairness*.

InvariantsandProperties

Invariants

$$
\begin{aligned}
\mathit{TypeOK} \triangleq\ & \forall\, p \in \mathit{ProcessID} : \\
& \land\ \mathit{State}[p].\mathit{ID} \in \mathit{ProcessID} \\
& \land\ \mathit{State}[p].\mathit{Condition} \in \{\text{“Active”},\ \text{“Dead”}\} \\
& \land\ \mathit{State}[p].\mathit{Leader} \in \mathit{ProcessID} \\
& \land\ \mathit{State}[p].\mathit{Participating} \in \text{BOOLEAN} \\
& \land\ \mathit{MessageBox} \in [\mathit{ProcessID} \to \mathit{Seq}(\mathit{Message})]
\end{aligned}
$$

$$
\mathit{UniqueID} \triangleq \forall\, p,\ q \in \mathit{ProcessID} : (\mathit{State}[p].\mathit{ID} = \mathit{State}[q].\mathit{ID}) \Rightarrow (p = q)
$$

Properties

$$
\begin{aligned}
\mathit{ElectionTerminationImpliesSameLeader} \triangleq\ & \forall\, p,\ q \in \mathit{ProcessID} : \\
& (\mathit{State}[p].\mathit{Condition} = \text{“Active”} \\
& \land\ \mathit{State}[q].\mathit{Condition} = \text{“Active”} \\
& \land\ \mathit{State}[p].\mathit{Participating} = \text{FALSE} \\
& \land\ \mathit{State}[q].\mathit{Participating} = \text{FALSE}) \\
& \Rightarrow (\mathit{State}[p].\mathit{Leader} = \mathit{State}[q].\mathit{Leader})
\end{aligned}
$$

Eventually, the election will end

$$
\mathit{ElectionWillEnd} \triangleq \forall\, p \in \mathit{ProcessID} : \mathit{State}[p].\mathit{Participating} \leadsto (\Diamond(\mathit{State}[p].\mathit{Participating} = \text{FALSE}))
$$

Figure 4.7: Invariants and Properties

TemporalFormula

$$
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{State},\, \mathit{MessageBox} \rangle} \land \mathrm{SF}_{\langle \mathit{State},\, \mathit{MessageBox} \rangle}(\mathit{Next})
$$

Figure 4.8: Temporal Formula

Chapter 5

Evaluation

5.1 The Model Checker

Both the Ring Algorithm and the Bully Algorithm have been thoroughly checked with the TLC Model Checker, which comes with the TLA$^+$ toolbox. The Ring Algorithm has been checked with $n = 1..10$, where n is the number of processes. The Bully Algorithm has been checked with $n = 1..5$. Both algorithms pass the checker without any errors, invariant violations or temporal property violations.

You can see the state space progress, with the diameter, number of states found and distinct states for the Ring Algorithm in Table 5.1 and for the Bully Algorithm in Table 5.2.

Ring Algorithm

Should the r^2 be said? I'm really unsure about this part in general

As you can see in table 5.1, the number of states found increase exponentially ($r^2 = 0.9964$).

Bully Algorithm

n	Diameter	States Found	Distinct States	Queue Size
1	1	1	1	0
2	3	3	3	0
3	9	17	13	0
4	19	66	38	0
5	27	232	101	0
6	40	773	262	0
7	53	2478	676	0
8	69	7710	1760	0
9	87	23434	4584	0
10	107	69923	11967	0

Table 5.1: Number of states found with the model checker in the Ring Algorithm

n	Diameter	States Found	Distinct States	Queue Size
1	1	1	1	0
2	3	3	3	0
3	7	50	28	0
4	14	7235	2628	0
5	29	7315267	2090268	0

Table 5.2: Number of states found with the model checker in the Bully Algorithm

5.2 Shortcomings

The biggest shortcoming, which has an actual effect on the algorithm is the Neighbour function (Figure 4.2), which does not accurately model a neighbour in a ring structure, as it assumes that only the leader can die. I decided against modeling the function in a more realistic way, as to focus on attempting to find out why the model would not run as expected.

5.3 Pseudocode to TLA$^+$

Instead of converting from pseudocode to TLA$^+$, I used a set of instructions for both algorithms. Both of these specified the behavior for when a process p wakes up or detects a leader to be dead. As we have assumed that processes that die are removed, there has been no need to model the waking up of a process, thus leaving us only with an option to create a function like *CheckLeader* (Figure 3.7.)

However, the approach of using a set of instructions like this, rather than a specific pseudo code has left a lot of ambiguity and things to be interpreted. While this may not seem the case at first glance, a few things aren't specified. Such as in the Ring Algorithm set of instructions (to be called pseudocode from here on), it is not specified when each process should toggle their *participation* status. Neither is it clear to see what the purpose of the *participation* variable is in general. The instruction set does not say at any point that something isn't allowed if a process isn't participating. This along with smaller things both from the Ring Algorithm and the Bully Algorithm has left a lot of interpretation work to do. Although it seems to me that the implementations of both algorithms should work, and follow the pseudocode in a way that makes sense, it does not work, and this might be the fault of too much interpretation work. It is specifically important that everything is modelled in TLA$^+$ as it explores every possibility, thus, if some edge case is not described in the pseudocode, and therefore hasn't been implemented, it might be the fault of the model checker not leading to a successful termination state.

Chapter 6

Conclusion

Missing

In this project we have implemented the Bully Algorithm and the Ring Algorithm for leader election. The algorithms have been implemented in the language TLA$^+$, which is a formal specification language used for designing, modelling, documentation and verification of programs.

Appendix A

Bully Algorithm TLA$^+$ model

Appendix B

Ring Algorithm TLA$^+$ model