Formal Verification

Kevin Joshua Vinther

May 17, 2023

1 Introduction

TODO: Confidence 1/10

Distributed Systems are very important...

A problem in Distributed Systems is the problem of leader election...

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 improtant to show some of the terminology.

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

2.2 Leader Election

TODO: Confidence 3/10

Leader Election is a type of algorithm in Distributed Systems, whose aim is to elect a leader from processes in a distributed systems. This is crucial for when a leader dies. The main challenge in leader election is that any process can die at any time, including the leader. Thus, if a leader dies, a new leader must be agreed upon by the other processes. Leader election assumptions:

- All nodes
- Fail-stop model

2.3 Bully Algorithm

TODO: Confidence 3/10

The Bully Algorithm is an algorithm that solves the Leader Election problem. In the bully algorithm, we assume a complete undirected graph.

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+

TODO: Confidence 5/10

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 they way the language works, by forcing you to write mathematically correct implementations, that can be tested in every possible state.

3 Bully Algorithm

3.1 Setup

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

The “setup” of the implementation consists of setting up the variables and constants we use, along with setting the initial values of the code.

$$\textsc{constant}\ N$$

$$\textsc{assume}\ N \in Nat$$

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

$$\textsc{variables}\ State,\ MessageBox$$

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

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

This code segment first defines a natural number constant, N, which chooses the number of processes in the system. The specific number N is first chosen in the modeling stage, where you check the model. We then model ID's of the processes by simply making a range from 1 to and including N, meaning that a process can have any natural number ID up including N.

The variables State and MessageBox are then declared, so that we can implement their initial values in the *init* function.

Before initializing, we declare that a message is the cross product between a process ID and its message "ELECTION", "ALIVE" or "VICTORY". This is used in the TypeOK invariant, to make sure that a message is a correct message.

In the *init* function we initialize the *State* and *MessageBox* variables. We set State to be a tuple of processes which map to a record of the meta-information of the process. The record which each process maps to has the fields ID, Condition, Leader and Participating. The ID is set to be the process ID, p. We set the condition to be "Active", because we want each process to be active in the initial state. The "condition" field could be set to be "Dead", in which case a leader election would immediately start. We set the Leader field to be the highest process, N. We also set the *Participating* field to be FALSE, as there is no election ongoing at the initial state. We initialize the MessageBox simply by mapping each process id to an empty tuple. Thus, we end with a tuple of tuples.

With this format, we can very simply access information from any process at the current state. The condition for example, can be accessed simply by evaluating *State[p].Condition*, where p is the process ID. Similarly, we can access messages from a process' message box easily with *(MessageBox[p])[1]* for the process ID of the sender, and *(MessageBox[p])[2]* for the contents of the message.

The ease of access here makes the rest of the implementation very readable.

3.2 Sending Messages

TODO: Confidence 4/10: General formatting, bad formulation etc.

```
SendAlive(p, q) ≜
    MessageBox' = [MessageBox EXCEPT ![q] = Append(MessageBox[q], ⟨p, "ALIVE"⟩)]


SendVictory(p) ≜
    ∧ MessageBox' = [q ∈ ProcessID ↦
            IF q ≠ p ∧ State[q].Condition = "Active"
             THEN Append(MessageBox[q], ⟨p, "VICTORY"⟩)
             ELSE  MessageBox[q]]
    ∧ State' = [State EXCEPT ![p].Leader = p]

ProcessHasTheHighestIDAlive(p) ≜ ∀ q ∈ HigherIDs(p) : State[q].Condition = "Dead"

SendElection(p) ≜ ∀ q ∈ HigherIDs(p) :
                        ∧ State[q].Condition = "Active"
                        ∧ MessageBox' =
                            [MessageBox EXCEPT ![q] = Append(MessageBox[q], ⟨p, "ELECTION"⟩)]
```

In general, with the way messages are modeled, we can send a message as simply as this:

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

This simply updates the MessageBox tuple for the receiving process, *q* with a message sent from *p* with the contents “MESASAGE”.

SendAlive
SendAlive sends a message from process *p* to process *q* with the contents “ALIVE”. The primary reason for this function is to help readability.

SendVictory
SendVictory sends a “VICTORY” message to every process with the condition “Active” which is not the process itself. The part that specifies that the receiving process should be active is not a necessity and could be left out without changing the functionality of the specification. The reason this is not done, is because it helps with a clean MessageBox. We can do this, because we assume **???? det der hvor processerne ved hvad der sker.** Furthermore, the process *p* which sends the message also sets its own leader to be itself, and its participation status to FALSE. Thus concluding it’s own election, by assigning a new leader and not participating anymore.

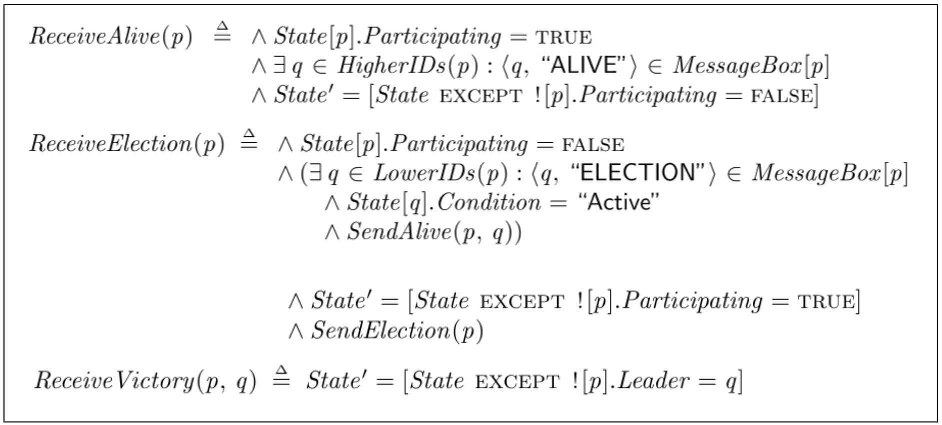
SendElection SendElection sends the “ELECTION” message to all processes with a higher id than the sender. To do this, it uses a helper function HigherIDs which is defined as such:

```
HigherIDs(p) == {q \in ProcessID: q > p}
```

Differences in the send functions
As one may think, these functions are quite different. The SendAlive function just sends a message without doing anything else. SendVictory sends a victory to every active process that is not itself, and SendElection sends only to higher process IDs. **TODO: Skriv hvorfor**

3.3 Receiving Messages

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



$$\begin{array}{ll}
\mathit{ReceiveAlive}(p) \triangleq & \land\ \mathit{State}[p].\mathit{Participating} = \text{TRUE} \\
 & \land\ \exists\, q \in \mathit{HigherIDs}(p) : \langle q,\ \text{“ALIVE”} \rangle \in \mathit{MessageBox}[p] \\
 & \land\ \mathit{State}' = [\mathit{State}\ \text{EXCEPT}\ ![p].\mathit{Participating} = \text{FALSE}]
\end{array}$$

$$\begin{array}{ll}
\mathit{ReceiveElection}(p) \triangleq & \land\ \mathit{State}[p].\mathit{Participating} = \text{FALSE} \\
 & \land\ (\exists\, q \in \mathit{LowerIDs}(p) : \langle q,\ \text{“ELECTION”} \rangle \in \mathit{MessageBox}[p] \\
 & \qquad \land\ \mathit{State}[q].\mathit{Condition} = \text{“Active”} \\
 & \qquad \land\ \mathit{SendAlive}(p,\ q)) \\
 & \\
 & \ \land\ \mathit{State}' = [\mathit{State}\ \text{EXCEPT}\ ![p].\mathit{Participating} = \text{TRUE}] \\
 & \ \land\ \mathit{SendElection}(p)
\end{array}$$

$$\mathit{ReceiveVictory}(p,\ q) \triangleq \mathit{State}' = [\mathit{State}\ \text{EXCEPT}\ ![p].\mathit{Leader} = q]$$

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 Handling Messages

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

$$
\begin{array}{ll}
\mathit{HandleMessages}(p) \triangleq & \land\ \mathit{State}[p].\mathit{Condition} = \text{“Active”} \\
 & \land\ \mathit{MessageBox}[p] \neq \langle\rangle \\
 & \land\ \lor\ \land\ \mathit{State}[\mathit{Head}(\mathit{MessageBox}[p])[1]].\mathit{Condition} = \text{“Dead”} \\
 & \qquad\ \land\ \mathit{MessageBox}' = [\mathit{MessageBox}\ \textsc{except}\ ![p] = \mathit{Tail}(\mathit{MessageBox}[p])] \\
 & \qquad\ \land\ \textsc{unchanged}\ \mathit{State} \\
 & \quad\ \lor\ \land\ \mathit{Head}(\mathit{MessageBox}[p])[2] = \text{“VICTORY”} \\
 & \qquad\ \land\ \mathit{ReceiveVictory}(p,\ \mathit{Head}(\mathit{MessageBox}[p])[1]) \\
 & \qquad\ \land\ \mathit{MessageBox}' = [\mathit{MessageBox}\ \textsc{except}\ ![p] = \mathit{Tail}(\mathit{MessageBox}[p])] \\
 & \quad\ \lor\ \land\ \mathit{Head}(\mathit{MessageBox}[p])[2] = \text{“ELECTION”} \\
 & \qquad\ \land\ \mathit{ReceiveElection}(p) \\
 & \qquad\ \land\ \mathit{MessageBox}' = [\mathit{MessageBox}\ \textsc{except}\ ![p] = \mathit{Tail}(\mathit{MessageBox}[p])] \\
 & \quad\ \lor\ \land\ \mathit{Head}(\mathit{MessageBox}[p])[2] = \text{“ALIVE”} \\
 & \qquad\ \land\ \mathit{ReceiveAlive}(p) \\
 & \qquad\ \land\ \mathit{MessageBox}' = [\mathit{MessageBox}\ \textsc{except}\ ![p] = \mathit{Tail}(\mathit{MessageBox}[p])]
\end{array}
$$

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.

$$
\begin{array}{ll}
\mathit{CheckLeader}(p) \triangleq & \land\ \mathit{State}[p].\mathit{Condition} \quad = \text{“Active”} \\
 & \land\ \mathit{State}[\mathit{State}[p].\mathit{Leader}].\mathit{Condition} = \text{“Dead”} \\
 & \land\ \mathit{State}[p].\mathit{Participating} = \textsc{false} \\
 & \land\ \textsc{if}\ \ \mathit{ProcessHasTheHighestIDAlive}(p) = \textsc{true} \\
 & \qquad \textsc{then}\ \ \mathit{SendVictory}(p) \\
 & \qquad \textsc{else}\ \ \mathit{SendElection}(p) \land \mathit{State}' = [\mathit{State}\ \textsc{except}\ ![p].\mathit{Participating} = \textsc{true}]
\end{array}
$$

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 **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.

$$\begin{aligned}
\mathit{KillLeader}(p) \triangleq\ & \land \mathit{State}[p].\mathit{Condition} = \text{“Active”}\\
& \land \mathit{State}[\mathit{State}[p].\mathit{Leader}].\mathit{Condition} = \text{“Active”}\\
& \land \mathit{State}' = [\mathit{State}\ \textsc{except}\ ![\mathit{State}[p].\mathit{Leader}].\mathit{Condition} = \text{“Dead”}]\\
& \land \textsc{unchanged}\ \mathit{MessageBox}
\end{aligned}$$

$$\begin{aligned}
\mathit{Next} \triangleq\ & \exists\, p \in \mathit{ProcessID} :\\
& \quad \lor \mathit{CheckLeader}(p)\\
& \quad \lor \mathit{HandleMessages}(p)\\
& \quad \lor \mathit{KillLeader}(p)
\end{aligned}$$

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.

3.7 Invariants and Properties

TODO: Confidence: 1/10, jeg har glemt at push nye variablenavne

To ensure that, while running the spec, we do not use insufficient or invalid data, we introduce invariants. One of these invariants is the TypeOK invariant, which acts as a sort of type checker for the program.

TypeOK

```
TypeOK == \A p \in Processes :
          /\ State[p].ID \in Processes
          /\ State[p].Status \in {"Active", "Dead"}
          /\ State[p].Leader \in Processes
          /\ State[p].ElectionInProgress \in BOOLEAN
          /\ MBox \in [Processes -> Seq(Messages)]
```

TypeOK makes sure that for each process, the ID, Status, Leader, Participating and Message Box is valid. It does this by making sure that the value is part of a predefined set (such as ProcessID being the set of processes.)

Other Invariants

4 Ring Algorithm

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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.

5 Evaluation

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

6 Conclusion

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.