RWTH Aachen University
Software Modelling and Verification Group

Verification of Leader Election Protocols for Dynamic Networks

Bachelor Thesis

presented by

Ján Jeremy Tugsbayar

1st Examiner: apl. Prof. Dr. rer. nat. Thomas Noll

2nd Examiner: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Advisor: Ira Justus Fesefeldt, M.Sc.

The present work was submitted to the Chair for Software Modeling and Verification

Aachen, September 18, 2025

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

Leader election is the process of designating a leader within a system with more power and responsibilities and it has many applications within a distributed system. The network of such distributed systems are often wireless and deployed in an environment which might not guarantee full reliability. This thesis focuses on leader election in unreliable networks, specifically its behavior, adaptations necessary to work successfully and viability. Using PRISM we model a network of nodes, that can face both node failures and communication failures between the nodes using a failure detection system. Then we introduce adapted leader election algorithms in the network that can work in conjunction with the failure detection. Using model checking we determine the viability of the leader election by determining the minimal and maximal chances of successfully electing a leader correctly in such a network.

Contents

Chapter 1

Introduction

In distributed systems, such as the modern Internet of Things (IoT) devices are often resource constrained and face unreliability. A good example are wireless sensor networks (WSN), which are battery powered, hence are resource constrained, and can experience unreliability. Proper coordination and cooperation is often necessary in such networks, one of the solutions is to have a specified coordinator or a so called leader. Leader election is a well known problem that is not deterministically solvable even under the assumption of perfect reliability. Some sort of asymmetry is needed to break the symmetry [San06], which then involves a randomized behavior.

Many leader election protocols exist for distributed systems, many of them assume a ring topology, for example the Ring algorithm [FL87] and LeLann's algorithm [LL77]. The ring topology often does not correspond to the real world layout of networks and would need a workaround or adaption similar to the routing algorithm Chord [JY06], which builds an overlay ring network and provides look-up and routing in the network based on the overlayed ring.

Unreliability is another challenge faced by distributed systems, be it unreliability in communication or outright hardware failure that disables a device. Many failure detection algorithms exist, but they are mostly standalone and work on a different layer of the network than the application or algorithm in question, such as leader election. An example for this is the widespread TCP protocol, which works together with the IP protocol and is able to detect packet losses and perform retransmissions. Such standalone protocols might not be suited for distributed systems of our interest, a tighter integration is required. The aim of this thesis is to combine a simple failure detection system in networks of distributed devices with adapted leader election algorithms, that can handle the problem of unreliability.

Suffice to say this is not a totally unexplored topic. As mentioned, many leader election algorithms exist and some are able to handle failures to a degree. The leader election algorithm by Chow, Luo & Newman-Wolfe [CLNW92] is able to detect the leader failing and trigger new leader election. The nodes have only local knowledge of the neighbors and the election is done by flooding the network by the neighbors of the leader that has failed. These neighbors are able to detect the leader failing, though communication failures are not assumed to be possible. The leader election algorithm in [KKKK95] preemptively elects a provisional leader and is also able to detect when the actual leader fails and switches to the provisional leader to maintain a leader. The Bully algorithm by Garcia-Molina [GM82] is able to handle network failures as well, though it is done by loosening the definitions of a correctly elected leader. Another algorithm from [GCLLR13] is adapted for mobile networks where devices can leave and join the network, hence nodes frequently

leave and join the network. The leader election in [ISWW09] is able to ensure a leader is elected in every connected component caused by topology changes. These existing algorithms are able to detect node or leader failures and changes in topology, which leads to communication links being disabled, but they do not take into account communication failures.

In the next Chapter 2, we will first introduce the concept of model checking and PRISM, a modeling language and model checking tool that was used in this thesis. PRISM was not the only tool used though, another tool called Storm was used, which proved to be a more efficient model checking tool for our use case that supports the PRISM modeling language. Chapter 3 gives an abstract overview of how we modeled the network. The network model can face both permanent node failures, and transient communication failures. To combat these problems we introduce a failure detection system and combine it with leader election algorithms. We perform model checking to observe the behavior and evaluate the feasibility of the leader election process, by analyzing the chances of a leader being elected successfully and correctly despite the unreliability of the network. Details of the implementation of the model will be discussed in Chapter 4. Chapter 5 deals with the evaluation of our results, lastly Chapter 6 concludes this thesis.

Chapter 2

Preliminaries

2.1 Model Checking

The complexity of designing and implementing algorithms keeps increasing with the increased digitization of our lives. To help avoid misbehavior while designing such algorithms **model checking** can be utilized. Systems and algorithms can be formally specified with a **formal model**, which describes their behavior. It is often discovered at later stages of design, that the system is inconsistent or incomplete. Model checking is a formal verification method with the aim of discovering these defects and the end-goal of proving that the model is correct. It performs this by exploring all the possible states that a formal model can have and by checking, whether a **specified property** holds [BK08]. Model checking has been successfully applied to various fields ranging from hardware verification [Fix08] to the health sector [BBD+06].

To perform model checking, the model has to be specified in a **modeling language**. These languages can vary from dedicated modeling languages such as Promela [Hol97] and Murphi [Dil70] to just regular programming languages used to implement a model. Additionally the properties that need to be verified must be defined. For this purpose **temporal logics** such as LTL, CTL are used. The model checker itself is a tool that computes all the possible states, including all the branches caused by various different choices or occurrences that can happen. Various model checking tools have been developed over the years and we will cover two specific ones that were used in this thesis.

2.2 PRISM

PRISM[1] [KNP02] is a model checker that supports **probabilistic behavior** and utilizes its own modeling state based language of the same name. PRISM has been widely used for modeling probabilistic behavior ranging from simple dice rolls [KY76] to biological cell processes [KNP08]. We have chosen PRISM as the modeling language for our model, since it allows us to model the unreliability of the network as a probabilistic behavior.

PRISM supports various model types such as discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and **Markov decision processes (MDPs)**. We have decided to use a MDP, as it features non-determinism. In an MDP model when there are multiple available choices, PRISM will choose an option non-deterministically. Additionally stochastic behavior is supported, which we utilize to model the unreliability of the network. In our model the stochastic behavior represents the choice of a message

[1] https://www.prismmodelchecker.org/

```
1  mdp
2
3  const double probability = 0.6;
4  const int max_value = 10;
5
6  global g : [0..2];
7
8  module node_A
9
10     x : bool init false;
11     [] x = false & y < max_value -> probability:(x'=true) + (1-probability):(x'=false);
12     [] x = true & y < max_value -> probability:(x'=false) + (1-probability):(x'=true);
13     [] y = max_value & g < 2 -> (g'=g+1);
14
15 endmodule
16
17 module node_B
18
19     y : [0..max_value];
20     [] x = true & y < max_value -> (y'=y+1);
21     [] y = max_value & g < 2 -> (g'=g+1);
22
23 endmodule
```

Listing 2.1: Example PRISM Model.

being successfully delivered or not and non-determinism is used when there are multiple choices to choose from. The stochastic behavior can be specified with a probability, which in turn represents the reliability of the network.

A simple example PRISM model is shown in Listing 2.1. **Modules** can be defined within PRISM with the keywords *module* and *endmodule*, these modules represent various entities or parts of a system. In our model, the individual nodes in the network are modeled as separate modules. In the example PRISM file, we can see two modules called *node_A* and *node_B*. Variables in PRISM can be **local or global** and available variable types are integers, double and Boolean. Integer and double variables have a defined value range in most cases, using the $[min..max]$ structure in the variable declaration. It is also possible to have unbounded variables for specific purposes such as approximate model checking. **Constants** have fixed value, that cannot be changed by the model during the model checking process. Local variables are declared inside a module and are tied to it, for example variables x on line 10 and y on line 19 are tied to modules *node_A* and *node_B* respectively. Modules can freely read and update their own local variables, but are unable to update local variables of other modules, they can only read their values, for example module *node_A* cannot change the value of the variable y of module *node_B*. Global variables and constants are declared outside of modules, In the example we have a global variable g and a constant *probability*. Global variables can be read and updated by any module. **Commands** describe the behavior of modules, they consist of a **guard** and an **update**. If the conditions of the guard are satisfied, the update can be performed, which changes the value of variables. There are three commands in the 2.1 at lines 11, 12, 13, 20 and 1.

Logical operators are also available in PRISM and the above mentioned command's guard uses a logical conjunction. Other types of operators are disjunction, equality, inequality and more. PRISM also supports formulas, which are expressions using logical operators, if-then-else statements and other standard logical functions. For convenience, there are also included built-in functions for rounding, minimum or maximum element from a given set. For a full comprehensive list of syntax and features such as labels, module renaming and synchronization of commands please refer to the PRISM manual[2].

In the example model in 2.1, node B increments the value of y until it reaches its maximum value of 10. Though it is only able to do it, while the Boolean variable x in module A is true. x alternates between true and false until y has reached its maximum

[2]https://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction

value with a probability of 60%. When probabilities are used in an update their sum of probabilities must be equal to 100%. In the command on line 11, the probability of x becoming true is 60% and the probability of it staying false is 40%, as indicated by the complement probability $(1 - probability)$ in the second part of the command. Afterwards, when y has reached the maximum value, both modules are able to increment the global variable g, but since both can, it is a non-deterministic choice left up to PRISM. The model checking works in steps, during each step, commands with satisfied guards must perform an update. These steps lead to different states of the model, i.e the possible configuration of all variables. All of the possible states form the state space of the model, which has to be fully built by PRISM to perform model checking.

To perform the actual model checking, a property has to be specified. The properties can be specified using various temporal logics, like PCTL, CSL and probabilistic LTL. Using the **property formula** $P\,max = ?\ [F\ g = 2]$ on the example model from 2.1 we can determine the maximum probability P of g eventually reaching the value of 2, which is 100%. But the minimal probability $P\,min = ?[F\ g = 2]$ of g reaching 2 is 0%, as in theory it is possible for x to stay false forever. In that case, y can in theory never reach the maximum value, hence g cannot incremented. Model checking with PRISM can be done on Windows or on Linux and both platforms support a GUI and a command line respectively a terminal version. A simulator tool that allows the user to select specific choices or let PRISM do random choices is also available. A property specification tool and model editing tools are also included in the GUI version. Using the simulator tool, if we let PRISM do random choices in the example model from Listing 2.1, it will use the included probabilities to make non-deterministic choices, so the amount of steps required until the model reaches the end can be different each time, because the variable x can stay false for a different amount of steps.

2.3 Storm

Storm[3] [HJK+22] is another probabilistic model checker, which supports various input languages such as PRISM, JANI and GSPNs. During the latter stages, we switched to using Storm for verification, as it was much faster in building and verifying the model. While PRISM model building has taken hours, Storm was able to perform it in mere minutes. This is achieved by Storm through various optimizations, such as state reduction and similar state merging.

Storm only offers a Linux terminal version, with which properties can be specified and verified, but does not offer a tool for model editing or simulations. PRISM model files can be directly processed by Storm, hence the transition over to Storm was effortless.

[3] https://www.stormchecker.org/index.html

Chapter 3

Abstract Model

This chapter will introduce our network and communication model, and leader election models on an abstract level. Some details and specifics will be left out, though we discuss the the details of implementation of the model in PRISM in Chapter 4.

3.1 Network

Our model is an MDP and utilizes probabilistic behavior to model the behavior of unreliability in the network. The network can be viewed as a **finite undirected graph** $G = (V, E)$ of n nodes forming the vertices and m amount links of the form by $E \subseteq (V \times V)$, which represent the **communication links** between nodes, which leads to a maximum amount of $\frac{n(n-1)}{2}$ edges. Edges are undirected, i.e symmetric, and anti-reflexive, so no self loops are allowed. Though for testing purpose we use significantly lower number of edges, so that it corresponds to a real life network topology.

The communication model is a **step-by-step** process without any concurrence, only a single node can be active at any given time. Step-by-step refers to the communication, once a node sends a message to its neighbor, it has to wait until the message is processed by the receiver. Meanwhile all other actions and every other node is blocked, so only one node is currently active at each time point. During modeling we faced severe hardware limitations due to sheer size of the state space caused by the complexity of the model, which lead us to vastly simplify the model at the cost of a higher level of abstraction and one of the consequences of simplification is this communication method. Moreover the communication is possible in both directions and is unbuffered. Nodes have an **ID**, which is used for identification. Each node possesses a **table of its direct neighbors** with their IDs. Using this table the nodes can choose to which node to send a message. Nodes can permanently fail, communication can also fail, but these failures are transient and only the single messages that are currently being transmitted are subject to failing, not the entire link. Note that a communication failure refers to the act of a message not being received at all or being received, but is too corrupted to be intelligible. In that case, the message will be discarded and this case can be reduced to the case of not receiving the message at all. Small transmission errors like a few bit flips can be handled by introducing redundant message coding and error correction within its capabilities, though this is out of the scope of this thesis and will not be covered or considered. Additionally, while it is possible to easily modify the model to enable failed nodes to recover to simulate temporary software crashes, that can recover themselves by simply rebooting or restoring to a previous state, we abstained from it to keep the model simple.

3.2 Leader Election

We decided to model two leader election algorithms. The first one is a modified version of the **Bully algorithm** by Garcia-Molina from [GM82]. The modifications were to fit into our network and communication model. The second one is Heartbeat Leader Election based on depth first search. First we will introduce the model of the original Bully algorithm under the assumption that the communication in the network is reliable, but nodes are able to fail. Afterwards our modified version of Bully. Lastly, we present the second leader election algorithm called **Heartbeat Leader Election** under the same assumptions of full communication reliability.

3.2.1 Original Bully Leader Election

The original Bully algorithm functions fully correctly only under some assumptions For us the most relevant ones are Assumptions 5, 8 and 9:

Assumption 5: If a node experiences a failure, the node halts all operations and some time later is able to resume from a previous state. Moreover, node failures cannot cause the node to behave unpredictably.

Assumption 8: Communication does not fail, i.e. if a message is sent, there is guarantee of it being received within a set amount of time.

Assumption 9: Nodes do not pause and respond quickly to messages. Failing to respond within a reasonable amount of time is considered a failure.

Under all these assumptions the Bully algorithm is able to guarantee that a leader will be elected. Each node has an assigned unique **identification number (ID)** with the values between 1 and n, with n being the number of nodes. The core concept of the algorithm is that a node with the highest ID bullies others into accepting it as the new leader. In the original Bully the leader has the role of assigning tasks to nodes. Failure of the leader can be discovered when there are no more new tasks assigned by the leader after a certain amount of time, which triggers a leader election. The algorithm can be split into **two parts**.

During the first part, a node that has detected that the leader has failed will start leader election and becomes the **candidate** for the election. It will attempt to discover whether there are other nodes with a higher ID by messaging every other node in the network. The other nodes answer back and if a node has a higher ID, the current candidate node will give up its bid in the election and simply wait until the leader is elected. The other node with a higher ID will be the new candidate and continues the election process. If there are other nodes with a higher ID but do not respond, then they have failed and the current candidate will become the leader.

In the second part, once the node with the highest ID becomes the leader and there are no functioning opponents left, the elected node needs to notify all other nodes that it has become the new leader. This process is performed by sending a message to all nodes to halt all current activities. Afterwards another message is sent to them informing them about the new leader. Now that all the nodes are aware of the leader, the nodes can resume in normal operation. It is possible for nodes to fail during the election process, if the winning candidate fails, the process is restarted after a certain amount of time. If a

node with a higher ID fails, for example the original leader, but recovers while the process is still ongoing, it will still participate normally.

3.2.2 Modified Bully Leader Election

Our modified version breaks Assumptions 5 and 8. Though Assumption 9 still holds, since the nodes respond immediately to incoming messages and there is no notion of time in our model. Another difference in our modified version is that the nodes generate an **election number** with values between 1 and n, with n being the number of nodes. These election numbers are used to compete in the leader election and are assigned to nodes randomly, but unique to each node. It is also possible to manually assign the elections numbers to specific nodes in order to setup specific conditions for verification. Though it is also possible to use node IDs as election numbers, similar to the original Bully algorithm, which grants a more predictable and stable election. The modified version also consists of two parts. The leader election can be triggered by the same method and the initiating node becomes the candidate. A difference in our model is that the node with the **lowest election number** does the bullying to become the leader, this is simply done out of convenience, as the second leader election algorithm was modeled first and we reused some parts of the PRISM model code. Moreover, since nodes are able to contact only their **direct neighbors**, the first part of the algorithm requires **multiple rounds**. Nodes can be marked as **visited** or **unvisited** in order to distinguish already contacted node. A simplified pseudocode of the algorithm can be seen in Algorithm 1

Part 1: The first part consists of nodes contacting its unvisited neighbors querying about their leader election number. In the pseudocode it is represented in the *while* loop on line 22. The receival of such messages is in the first if branch. The difference between the original Bully and ours is that we cannot directly contact all the other nodes in the network, henceforth the candidate will contact all of its direct neighbors with a message informing them about the leader election, its own election number and querying the neighbor about their election number. The neighbors will receive these messages and compare the election number of the candidate with their own election number, afterwards the neighbors respond back with a reply with their own election number. If a neighbor has a lower election number, the current candidate node will know this based on the reply and will give up its candidate status. That neighbor with the lower election number will become the new candidate and the previous candidate will set the new candidate as its leader for now. If none of the neighbors have a lower election number, the current node will keep its candidate status and the neighbor will set the current candidate as their leader.

Because we can only contact direct neighbors, the algorithm works in rounds. After contacting all of its neighbors, the current node is marked as finished and a new round begins. In the new round, the neighbors that were contacted in the last round, now proceed to contact its own neighbors, but excluding the ones that were already contacted.

Part 2: This part is activated only if a neighbor with a lower ID was discovered and it becomes the new candidate, in the pseudocode it is represented by the *else* branch starting at line 14 of the first if. When a new candidate is chosen, the original candidate has the responsibility of informing its neighbors about the candidate change. This includes all the previous nodes that are not direct neighbors of the original candidate. To perform this backtracking is initiated. In backtracking the information about the new candidate is sent in a backwards direction in order to reach all the previous nodes.

If the current candidate fails, the election process is restarted again. If a participating

Algorithm 1: Modified Bully Algorithm Pseudocode

```
   Data: Table of neighbors with visited or unvisited status
 1 int id;
 2 bool visited;
 3 int election_number;
 4 int leader = election_number;
   /* When we receive a message, the value of election number from it
      is stored here                                                  */
 5 int incoming_message_content;
   /* Part 1, this node was not visited                               */
 6 if visited = false then
 7     if incoming_message_content < leader then
           /* When the challenger has a lower value leader           */
 8         leader = incoming_message_content;
 9         visited = true;
           /* Part 2, candidate change                               */
10         send message to all visited neighbors with int leader;
11     else
           /* When the challenger has a higher value leader          */
12         visited = true;
13     end
14 else
       /* Part 2:  Backtracking                                       */
15     if incoming_message_content < leader then
           /* When the challenger has a lower value leader           */
16         leader = incoming_message_content;
17         send message to all visited neighbors with int leader;
18     else
           /* When the challenger has a higher value leader          */
19         nothing
20     end
21 end
22 while visited = true and unvisited neighbors exists do
23     send message to unvisited neighbor with int leader and mark the neighbor as
        visited;
24 end
```

non-candidate node fails, it should not hinder the leader election in any way unless it makes the network disjoint. However, when we have to consider communication failures, we need additional steps. This part will be covered in the part about integration of failure detection with the leader election in Section 3.3.3.

3.2.3 Heartbeat Leader Election

Heartbeat Leader Election is an algorithm of our own design and is based on the depth first search (DFS) algorithm [Erc13]. To the best of our knowledge, we could not determine whether an leader election algorithm like this has been done before. From now the term heartbeat will be shortened as HB when appropriate. The pseudocode for HB LE is provided in 2.

Analogue to Bully, nodes generate an election number, which is then used for the leader election. It is also possible to simply use the node IDs for leader and leader election can be simply triggered by a node that detects that the leader is not responding anymore, same as in Bully. When the leader election is triggered, the initiating node becomes the **candidate**. For the heartbeat to work the nodes are marked either as visited or unvisited. Once the HB leader election is initiated by a node, it chooses a **single unvisited neighbor with lowest ID** to **visit**. Visiting simply means sending a message informing the neighbor that the leader election is currently in progress and includes the candidate's election number, and asks for a response. If the neighbor's election number is lower, the neighbor responds back stating that the neighbor has become the new candidate. On the contrary, if the neighbor's election number is higher, the neighbor responds to the current candidate with an acknowledgment reply, stating that it accepts the candidate. In both cases, the neighbor then repeats this process of visiting a further unvisited neighbor with the lowest ID.

Following this procedure, we create a **path** through the network. Depending on the topology of the network, we may get stuck at a **dead end**, i.e the current node has no unvisited neighbors. To alleviate this, we use backtracking similar to DFS. When there are no unvisited neighbors available, we **backtrack** to the previous node by sending a backtracking message. At the previous node we then check again for unvisited neighbors. If there are again no unvisited neighbors, we backtrack again. If there are unvisited neighbors we simply continue with the regular algorithm by visiting that neighbor. Once all nodes were visited, backtracking will be performed all the way back to the initiator of the leader election. This ensures that all the nodes in the network are aware of the newly elected leader.

It is possible that a new leader will be found in the later stages of the algorithm. To ensure that all nodes will be aware of the leader, once we have visited all the available nodes, we perform the same process in the other direction starting from the node where we ended. With this method we should visit every node again and that ensures that every node is aware of the correct leader.

3.3 Failures

In the previous section, we introduced two leader election algorithms under the assumption that nodes are able to fail, but the communication is reliable. Now we introduce a simple mechanic for failure detection and recovery.

Algorithm 2: Heartbeat Leader Election Pseudocode

```
   Data: Table of neighbors with ID and with visited or unvisited status
 1 int id;
 2 bool visited;
 3 int election_number;
 4 int leader = election_number;
   /* When we receive a message, the value of election number from it
      is stored here                                                    */
 5 int incoming_message_content;
 6 bool backtrack_to_initiator;
 7 if visited = false then
 8     if incoming_message_content < leader then
           /* When the challenger has a lower value leader             */
 9         leader = incoming_message_content;
10         visited = true;
11         send a reply back;
12         if an unvisited neighbor exists then
13             send a message to unvisited neighbor with lowest ID with int leader
14         else
15             backtrack;
16         end
17     else
           /* When the challenger has a higher value leader            */
18         send a reply back with int leader;
19         if an unvisited neighbor exists then
20             send a message to unvisited neighbor with lowest ID with int leader
21         else
22             backtrack;
23         end
24     end
25 else
       /* Backtracking                                                 */
26     if incoming_message_content ≤ leader then
           /* When the challenger has a lower value leader             */
27         leader = incoming_message_content;
28         send a reply back with int leader;
29         if an unvisited neighbor exists then
30             send a message to unvisited neighbor with lowest ID with int leader
31         else
32             backtrack to the most recently visited neighbor with a message
                containing int leader;
33         end
34     else
           /* When the challenger has a higher value leader            */
35         send a reply back;
36         if an unvisited neighbor exists then
37             send a message to unvisited neighbor with lowest ID with int leader
38         else
39             backtrack to the most recently visited neighbor with a message
                containing int leader;
40         end
41     end
42 end
```

3.3.1 Failure Detection

In both leader election algorithms, nodes message other nodes with a query and a reply should follow. There are three possible scenarios of failure that we consider:

1. Node A sends a message to node B, but node B has not received it, so a reply will not be sent.
2. Node A sends a message to node B, node B receives it successfully and sends a reply message. This reply message is then not received by node A.
3. Node A sends a message to node B, but node B has failed and has not received it, so a reply cannot be sent.

Whenever one of the above three cases of failure occurs, the sender of the message (node A in the scenarios) will enter a **recovery mode**, which will attempt to reestablish communication. In our model the failure is discovered instantly, in reality a timeout would be used to detect lost messages. Implementing a timer in our MDP model would drastically increase the complexity, by creating new states with the only difference being a time variable, hence it was omitted.

3.3.2 Recovery

The nodes store information from which neighbor they received a message and to which neighbor they sent a message. When a recovery is triggered by any of the three cases above Section 3.3.1, the node does not know whether a link or node failure has happened. In recovery mode the node will attempt to send **recovery messages** to the affected node. The number of allowed attempts to reach the affected node can be adjusted in the model.

In case of just a communication failure, it is then possible that the recovery message will reach the affected node. Upon successfully receiving a recovery message, the affected node then proceeds to send back a **successful recovery message** that the recovery was successful and we exit recovery mode.

If the affected node has failed, we will eventually reach the maximum amount of recovery attempts. Similarly, if the node has not failed but the communication is just so unreliable that none of the recovery messages have reached the affected node, we will also reach the maximum amount of attempts. In both cases, the node will exit recovery mode and will proceed to ignore the affected node for the foreseeable future.

3.3.3 Integration with Leader Election

In both leader election algorithms, nodes send messages with queries that require a reply and some that are just informative. Now we enforce the rule that every message requires a confirmation reply. By enforcing this rule we enable our failure detection to function properly. Moreover, it actually reduces the three cases of possible failures from Section 3.3.1 into a single case: No reply was received after sending a message. If the node we are trying to contact has failed, no reply will ever come. If the node we are contacting has not failed and did receive our message, but we could not receive the reply, this also leads to the same result, hence the recovery response will always be the same.

For both leader election algorithms if a node A fails to contact node B, node A will simply ignore node B. If node B has a different neighbor, for example node C and the election progresses to node C, it will still try to contact node B. The failure detection

scenario was reduced to a single. Though for cause of the failure we consider three possible scenarios that can happen when a certain node is affected by failure:

1. The affected node has failed, none of its neighbors are able to contact it, hence it will be excluded from leader election.

2. The affected node has not failed, but none of its neighbors are able to contact it. This can be reduced to case one.

3. The affected node has not failed, some of its neighbors were not able to contact it, but some other could. In this case the node can participate normally.

In the end we, only have two true outcomes. The recovery was successful and the node will **participate** in the leader election or the recovery was not successful and the node will be **excluded**. In our modified Bully, if a neighbor does not respond and the recovery has failed, the current node will simply ignore the affected node for the duration of the leader election. If the affected node has different neighbors, case three is still possible.

In Heartbeat Leader election, if a recovery has failed, the current node will simply choose a **different unvisited node** to visit if there are any available. If there are no unvisited nodes available, backtracking is initiated. For better integration and increased robustness we introduce a round mechanic into HB Leader Election. If a failure occurs which causes the algorithm to be not able to continue, e.g. no unvisited neighbors which we have not already tried to visit and failed or no neighbors to backtrack to, the algorithm will restart itself with a new round. To make it no loop infinitely a maximum number of rounds can be set. If the failure occurs during backtracking.

Chapter 4

Modeling in PRISM

This chapter gives an insight into the PRISM model we have implemented and serves as an introductory documentation of the source code. The source code itself is provided separately. We introduce the problems we faced while modeling in PRISM, which forced us to make certain decisions in the model and the justifications are provided.

4.1 Abstraction and Simplification

As alluded before, during the implementation of the model we have faced problems with the size of the state-space of the model and were forced to take some deliberate **design choices** and switch tools. At one point the model had 20 million states while not even being finished yet. At first we strived to build a model that was as close of a match to the reality of how a network behaved. However, due to limitations a much higher level of abstraction from reality had to be inserted.

Initially all the verification was done using PRISM model checker tool, but as the model expanded it became unpractical to use PRISM further due to a state-space explosion and memory requirements. To alleviate this resource problem, we tried to use powerful remote compute clusters. Even on the clusters the model would not build within 24 hours using PRISM. PRISM itself is implemented using Java and if too much RAM was assigned, Java exceptions were thrown. This has lead to several rounds of **simplification** of the model, some features such as a link specific reliability values, full disabling of links after failure and concurrent communication between multiple nodes were cut in order to reduce the number of states.

Halfway through the implementation of the model, we decided to switch to Storm for the model building and verification part. Storm performed admirably, it was possible to build and verify the model within just a few minutes even on personal computers. However, the switch occurred when the entire model was not finished yet. Over time as more work was done, Storm also required hours to build the model. We had to keep these limitations in mind when implementing the model further.

During the simplification phase many of the local variables were changed to global in order to reduce the number of states caused by the local variables of each node. That has unfortunate effect of the model being able to **“cheat”**. Nodes were able to perform decisions based on global variables, which are accessible to all the nodes, whereas in reality the information would have to be passed on as messages between nodes. The biggest consequence of this is visible during the **termination** of leader election, where a separate module called the **controller** is able to access information about the state of

all nodes in order to decide whether to terminate or not. A separate central controller with access to all the information is not doable in reality in a distributed leader election system, but for us to work efficiently it was needed as an **optimization and abstraction** step. A solution to this would be to include information about the leaders of all visited nodes and their current set leaders, but that would increase our state-space with all the possible permutations of the current leaders of all nodes. These are naturally undesirable consequences that causes the model to not reflect the reality. On the other hand we avoided 12 hour long model building process after each development iteration.

4.2 Network and Communication Model

The nodes in the network model represent the devices in the network. Below, you will find in Listing 4.1 a simplified model of two nodes that can send each other a single message. The nodes are given letters as identifiers, such as node A and node B. These letters serve only for distinguishability in the PRISM code. For the purposes of the algorithms in the model, nodes are additionally given an unique integer based local variable called **ID**, for example A_id is the ID for node A. Local variables contain a letter prefix, which indicates to which node it belongs, such as A_status. The nodes have a single data variable A_data and B_data respectively.

```
1  mdp
2
3  const double global_send_reliab = 0.7;
4  //Probability of a message being successfully received.
5
6  global a_msg : [0..2] init 1;
7  global b_msg : [0..2] init 1;
8  //Incoming message indicator.
9
10 global ab : bool init true;
11 //Edges
12
13 global msg_content : [0..10] init 0;
14 global msg_type : [0..4] init 0;
15 //Message specifics.
16
17 module node_A
18     A_id : [0..2] init 1;
19     //ID.
20     A_data : [0..10] init 2;
21     //Local data storage.
22     A_status : [0..3] init 1;
23     //Status
24
25     [] A_status = 1 & ab = true & msg_type = 0 & msg_content = 0 & b_msg = 1 -> (b_msg'=2) & (msg_type'=1)
           & (msg_content'=A_data) & (A_status'=2);
26     //Sending a message to B.
27     [] A_status = 1 & a_msg = 2 -> global_send_reliab:(a_msg'=1) & (A_data'=msg_content) & (msg_type'=0) &
           (msg_content'=0) & (A_status'=3) + (1-global_send_reliab): (a_msg'=0) & (msg_type'=0) &
           (msg_content'=0);
28     //Receiving a message from B.
29     //70% chance of the message being successfuly received by setting a_msg = 1 and a 30% of if failing by
           setting a_msg = 0.
30 endmodule
31
32 module node_B
33     //Same layout as A, but variable prefix is changed.
34     B_id : [0..2] init 2;
35     B_data : [0..10] init 4;
36     B_status : [0..4] init 1;
```

Listing 4.1: Example PRISM Model.

To represent communication and messages between the nodes, the global variables msg_type and $msg\_content$ are used. Additionally each node also has a incoming message variable, in the listing 4.1 these are a_msg and b_msg, which indicates an incoming message to be processed. Global Boolean variables like ab are used to represent existing communication channels between nodes, which can simply be called links. In the case of ab, it indicates the link between node A and B with the value true meaning the link

exists. It is important to note, that in order to reduce superfluous variables the variable *ab* represents the link in both direction. Adjusting the values of these variables in a larger network allows easier verification of different network layouts. Based on the link variables, the nodes are able to contact specific neighbors. Node A is able to send a message to node B by setting the variables *b_msg* to 2 *msg_type* to 1. The value 1 for *msg_type* indicates a regular message. The value of 0 means that no message is currently being sent and for other types of messages such as leader election messages or recovery messages different values are used for distinguishment. Node A is only able to do it when the channel *b_msg* is unused, i.e. *b_msg* has the value of 1 and *msg_type* is 0. The value 0 for an incoming message variable like *b_msg* indicates a failure to receive the message. It is also possible to introduce a payload of data using the variable *msg_content*, in the Listing 4.1 the payload is the value of A_data. Once a message is sent (line 22), node B is able to detect and process the message based on the variable *b_msg* in line 34 and store the payload into its own data storage. Either of the two nodes is able to send a message at the start, but once a single message is sent and received, the model terminates, because the status variables in both nodes changes. Using the status variables *A_status* and *B_status* allows us to control the action and response flow of nodes.

Nodes and message transmissions are given a **reliability** value. The message can fail to be received, the chance of that is dictated by the variable *global_send_reliab*. The chance of failing is 30% and when it occurs, the message variables *a_msg* or *b_msg* get set to 0, the same applies to other message variable for message type and content/A communication reliability of 0.7 means that there is a 30% chance of a failure occurring and 70% chance of the transmission to be received successfully.

Analogously nodes perform periodic **failure checks**, during which they can fail. Node failures represent hardware failure within nodes that disable them permanently. Communication failures occur to single messages, but both the sender and receiver are unaware whether a failure has happened or not. To alleviate this, we employ a failure detection system. This forms the basis of our communication model.

4.3 Modeling Failures in the Network

4.3.1 Failure Chances

Because of our communication model, sending a message is always successful but receiving it may not be. When nodes receive a message, the variable *global_send_reliab* determines whether it was received successfully or not. To implement node failures, we use the variable *global_node_reliab*. Each time a node receives a message successfully, all of the nodes in the network, except for the current node that has just received a message, perform a fail check. The fail check uses the variable *global_node_reliab* to determine for individual nodes whether they will fail now or not fail. Originally all of the nodes had their own specific message receiving failure and node failure values, but that feature was cut, hence the variables have a *global* prefix.

An interesting quirk of our implementation is that, originally a different reliability variable *global_reply_reliab* was used for the reply messages. But as explained in Section 3.3.3, all failures can be reduced to the simple case of a reply not being received. In order to avoid modifying a lot of code in the model, we have decided to permanently set the value of *global_reply_reliab* to 1.0, meaning replies are always successful. The recovery response for all three cases is the same, so the nodes are not able to distinguish between a reply message failing, the original message failing or no reply incoming because the node has failed.

The chance of a reply failing is now "included" in the *global_node_reliab* variable, which affects whether the original message to the node will be received or not. The other reason for this choice, is that it reduces the number of states in the model, which significantly reduces the build time of the model.

Lastly, the variable *global_recov_reliab* dictates the reliability of recovery messages. In all of our testing and verification the *global_recov_reliab*, is set to a higher value than *global_send_reliab*, the reasoning behind it being that recovery messages may contain less data and are simpler then regular messages or a different method is used, such as a separate frequency in case of wireless networks so it has a higher chance of being successfully received. If the affected node itself has failed, all of the incoming messages will fail to be received.

4.3.2 Recovery

As explained in Section 3.3.1 and shown in the communication model in 4.1, nodes can **detect instantly** whether a sent message was received successfully or not. This is done using the node specific incoming messages variables, for example a_msg for node A. If the message was not received, its value will be set to 0 and this alerts the sender of the message. After detecting a failure, the sender of the message will reset the value of a_msg back to its default value of 1. Then the recovery process begins.

Nodes have a variable that contains the ID of the receiver of an outgoing message. When recovery is initiated, its value will be copied over into the variable $A\_recovery\_id$. This is then used to distinguish which neighbor is the affected node of failure. After establishing this information, nodes will proceed to send recovery messages to the affected node. The amount of how many recovery attempts are allowed depends on the variable $max\_recovery\_attempts$. Tweaking this variable together with the reliability values allows us to set up various characteristic of the network, for example failure heavy but easily recoverable or failure light and not easily recoverable. If the affected node has not failed, but experienced only a communication failure, it may receive the recovery message. Upon receiving it, it will respond back with a **successful recovery message**, that functions as a reply to conform to the communication model. With similar reasoning to reply reliability, we decided to make this successful recovery messages 100% reliable in order to reduce the number of states.

If there is an ongoing leader election and a recovery was successful, the leader election will continue normally. How each leader election algorithms deals with unsuccessful recovery will be covered in their specific sections.

4.4 Leader Election

4.4.1 Bully Leader Election

In our Bully leader algorithm the variable *active_node* indicates which node is currently in part one of the algorithm. During part one, nodes have a choice of which neighbor to contact. This choice is left for the model checker to perform and is non-deterministic. However, when a new round is started, due to our model not supporting concurrency, only a single node is chosen to be active and the variable *active_node* will change. The new active node, will be one the neighbors of the previously active node, that was contacted. Specifically it will be the neighbor with the lowest ID. If there was only a single neighbor that was contacted, then there is obviously only one choice.

Each node also has an assigned mark variable, for example *A_mark* for node *A*. The mark variables are global and are used in the process of selecting the new active node when a new round begins, it is also used to determine which nodes to contact and which to not. This is one of the examples of "cheating" mentioned in Section 4.1. Nodes can make decisions using global variables, instead of relying on their local knowledge. In a situation of 3 nodes labeled *A*, *B* and *C* in a triangular network. If node *A* messages *B* and *C*, but *C* fails to reply and recovery failed. Node *B* will attempt to contact *C* based on the global variable *C_mark*. In reality this information would have to be communicated between nodes via messages and stored locally, but in our model it is immediately available to all of the nodes. Messages have a single payload variable *msg_content* and that is reserved for sending the leader election number for comparison. Supporting more data in messages, such as failed contact attempts, would introduce more states that we want to avoid.

The initiator of the leader election is chosen randomly, though it is possible to set a specific node to initiate for testing purposes. To perform this we use a variable *temp_leader*, a random value in the range of node IDs is assigned to it, then the node whose ID matches the value of the variable initiates the leader election. Leader election would be normally triggered if the current leader fails, though this mechanic is not included in our implementation in order to reduce the number of states.

The variable *election_phase* indicates in what phase the election currently is. This variable is used to enter part two of our version of Bully. During termination of the leader election we observe another unrealistic feature as consequence of simplification. In Section 4.1 we introduced the need for a controller module. The controller is a separate module in the model that is able to determine when we enter different *election_phase* and when the leader election terminates. The controller can see the leader variables of all the nodes and is responsible for the termination of the algorithm. Naturally a centralized controller does not fit into a distributed system. Without the controller the nodes would require another round of leader confirmation messages, in order to make sure that every node is aware of the correct leader. This would increase the state-space of the model, which we want to avoid to reduce the build time of the model.

It is possible that a node failure or an unrecovered communication failure causes the network to be disjointed. There may be unreachable nodes that have not failed, because the connecting nodes in-between have failed or could not be reached. The algorithm makes decisions based on local node knowledge, so it may not even be aware of unreachable nodes. In this case the algorithm simply terminates in the current part of the split network, where it is currently active. The termination is done by the controller, which can see that some nodes are unreachable, which is another undesired aspect, but necessary for optimization.

4.4.2 Heartbeat Leader Election

To initiate leader election a random node is chosen non-deterministically with the same variable *temp_leader* as in Bully. The chosen node then starts the process of neighbor calculation. The neighbor with lowest ID is calculated using a formula and is stored in the variable *A_lowest_neighbor*. Afterwards the node proceeds to send a message to that neighbor. If there are no unvisited neighbors, we do regular backtracking to the most recently visited node.

By visiting the neighbors with the lowest ID and backtracking when necessary, the algorithm should visit all reachable nodes. Analogue to Bully, nodes have a mark variable. Each time an unvisited node is visited, it is marked as visited. To enable backtracking, there is a global variable *visit_counter* and local node variable *A_visit_order*. The variable

visit_counter is incremented after each new visit and nodes assign themselves a number representing the visit order in *A_visit_order*. This visit order represents the path the algorithm has chosen. For the process of backtracking the nodes choose a visited neighbor which has the highest *A_visit_order* value, i.e. the most recently visited neighbor.

In addition to the mark variable of nodes, nodes also have a variable $A\_finished$. When a node decides that it has no unvisited neighbors, it will initiate backtracking and set itself as finished. Backtracking is done to nodes which do not have the finished status. The algorithm terminates when there are no unvisited or unfinished neighbors available.

Once all the nodes are visited and the leaders of nodes are not set, the next round begins, which starts from the node, that we have ended on. It is possible that the leaders of all nodes are already set, this can happen for example when the initiator of the leader election is the node with the lowest election number. In this case the already correct leader information is being sent to every node since the start. When a new round begins, the variables *election_reset* is used to reset all the local and global variables to their starting values except for the IDs and leader variable of nodes. Without any failures occurring, a single or two rounds are enough to make sure that all nodes have the correct leader. Though when taking failures into consideration more rounds may be necessary. The variable *max_le_rounds* sets the maximum amount of rounds. New round of HB are started when the algorithm is stuck. This can happen if a failure occurred and recovery has failed, as nodes do not attempt to contact nodes again, with which the recovery has failed. However during the reset process, the nodes forget with which neighbors a recovery has failed in the previous round.

Similar to Bully if the network gets disjointed, the HB Leader election will stay in the partition, in which the currently active node is and perform the leader election there. The cut off nodes in the other partition are deemed as failed. The algorithm terminates successfully when we backtrack to the initiator of the leader election, though we must also consider the case that we backtracked through the initiator during part one. A solution to that is checking whether the correct leaders are set on all visited nodes when we reach the initiator, this is done by the controller is, as the nodes themselves cannot directly determine this.

Chapter 5

Evaluation

To verify our leader election algorithms component-wise. First we verify the correctness of only the leader election, then with only a single types of failure and lastly with all types of failure enabled.

To verify properties we first need to define what a successful leader election is. We differentiate between a **weak success** and **strong success**. In weak success, if it was not possible to contact a node because of unrecoverable communication failure, but the node has not failed, it is deemed as a success. In the other definition of a strong success, when a node could not be contacted because of unrecoverable communication failure and the node has not failed, we consider it a failure. Another aspect that requires a definition is the **correct minimal leader**. In both weak and strong success, the correct leader is the node with the **lowest ID that has not failed**. The goal of the leader election is to have all nodes having the correct minimal leader. The properties that we use for verification is the termination of leader election with weak success or strong success.

Different states of the model can be defined using a label. We have a few defined labels that represent the different outcomes of the leader election, such as different types of success or failure. Temporal logics such as pLTL or pCTL can be used to determine the probability of labels being reached and holding true or not. To represent this probability two metrics are used: $Pmin$ and $Pmax$. The metrics represent the minimum and maximum chance of the property holding based on the decisions made by the model checker.

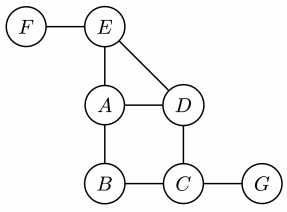

Figure 5.1: Seven node network.

All the results of the verification provided is performed on a 7 node network shown in Figure 5.1. Verification on other network topologies were carried out, but we decided that this type of a topology is a good real world example and provides the most interesting result. **Leader election numbers** that are used for competing in leader election are based on the **IDs of the nodes**. The nodes have a letter identification to be distinguishable and the IDs are assigned in an incrementing alphabetic order, e.g. node A has the ID value 1 and node B the value 2. In Appendix A we give exact instructions on how to replicate the result of our verification and clarify the formulas behind them.

5.1 Bully Leader Election

To verify our modified Bully leader election we verified 7 properties. Unless specified, the node that starts the leader election is random. The random starting node is a non-deterministic choice made by PRISM.

Bully Property 1: Correctness of leader election. Network configuration:

- $global\_send\_reliab = 1$
- $global\_recov\_reliab = 1$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$

We first verify that our modified Bully leader election algorithm is correct under the assumption that no failures are allowed. We perform this by verifying whether all of the nodes have the correct leader. Since there are no failures, all of the nodes will be reached. Hence the correct leader is node A with the election number 1. Both $Pmin$ and $Pmax$ are 100%, which means that the specified property of every node having the correct minimal leader holds in every case.

Bully Property 2: Correctness of the failure detection and recovery.
Network configuration:

- $global\_send\_reliab = 0.7$
- $global\_recov\_reliab = 1$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$.

Now we verify that our failure detection and recovery procedures are correct. To do this we allow communication failures to occur but make the recovery messages fully reliable. It does not matter what the reliability of the non-recovery messages are, they can be anything in the range (0, 1). Nonetheless we get $Pmin = 100\%$ and $Pmax = 100\%$, which shows that our failure detection and recovery does perform correctly and the leader election is able to recover from communication failures.

Bully Property 3.1: Weak leader election success with only communication failures, high reliability and low amounts of recovery attempts. Network configuration:

- $global\_send\_reliab = 0.8$
- $global\_recov\_reliab = 0.9$
- $global\_node\_reliab = 1$

- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$

In this case we allow full communication failures, but keep the reliability values high and we allow only 2 recovery attempts. It is possible then, that some nodes will not be reached. We allow a maximum of 2 recovery attempts. When we have the model to choose non-deterministically a random starting node we get $Pmin = 98\%$ and $Pmax = 100\%$. The $Pmax$ value intrigued us and we investigated it further by verifying the same property individually for each starting node. This was done setting a single starting node. The 100% maximum chance occurs when the starting nodes are A or B. These are coincidentally the nodes with the 2 lowest leader election numbers. The $Pmax$ values from other starting nodes are between 77% and 92%. Each different starting node is a different branch in the model and when we verify with a random starting node, the value $Pmax$ gives the highest value of all choices so that is why the end result is 100%.

Bully Property 3.2: Weak leader election success with only communication failures, low reliability and low amount of recovery attempts. Network configuration:

- $global\_send\_reliab = 0.4$
- $global\_recov\_reliab = 0.6$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$

If we lower the reliability values even further, but do not increase the maximum allowed recovery attempts, our chances drop to $Pmin = 49\%$ and $Pmax = 100\%$. For the $Pmax = 100\%$ the same argument applies as in Property 3.2.

Bully Property 3.3: Weak leader election success election with only communication failures, low reliability and high amount of recovery attempts. Network configuration:

- $global\_send\_reliab = 0.4$
- $global\_recov\_reliab = 0.6$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 5$

Now we increase the maximum amount of recovery attempts. If we allow up to 5 recovery attempts, our chance of success rises back up to $Pmin = 95\%$ and $Pmax = 100\%$. It proves that, it is possible to cope with network unreliability by increasing maximum allowed recovery attempts.

Bully Property 4.1: Strong leader election success election with only communication failures, high reliability and low amounts of recovery attempts. Network configuration is the same as in Property 3.1.

With a more strict definition of leader election success we get slightly lower values of $Pmin = 98\%$ and $Pmax = 99\%$ and we do not get the 100% chance of maximum success anymore.

Bully Property 4.2: Strong leader election success with only communication failures, low reliability and low amounts of recovery attempts. Network configuration is the same as in Property 3.2.

Similar to Property 4.1 we get lower values compared to weak success leader election. The verification results are $Pmin = 40\%$ and $Pmax = 75\%$

Bully Property 4.3: Strong leader election success with only communication failures, low reliability and high amounts of recovery attempts. Network configuration is the same as in Property 3.3.

As in weak leader election, we can significantly boost our success chance by increasing the amount of allowed recovery attempts, so we get the values $Pmin = 94\%$ and $Pmax = 98\%$.

Bully Property 5.1: Weak leader election success with all types of failures and low node reliability. Network configuration:

- $global\_send\_reliab = 0.7$
- $global\_recov\_reliab = 0.8$
- $global\_node\_reliab = 0.7$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 3$

Nodes can now fail, which may leave some nodes unreachable that have not failed. For this case only the weak success can be applied, as we can still elect the correct leader in the reachable nodes. With all types of failures enabled our minimum success chance drops further to $Pmin = 44\%$ while the maximum success chance stays at $Pmax = 100\%$ like in other cases of weak leader election successes. We tried increasing the amount of recovery attempts, but that gave us only a marginal improvement of 1%.

Property 5.2: Successful weak leader election with all types of failures and high node reliability. Network configuration:

- $global\_send\_reliab = 0.7$
- $global\_recov\_reliab = 0.8$
- $global\_node\_reliab = 0.95$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 3$

Since increasing the amount of recovery attempts did not boost our chances, we decided to increase the node reliability to a higher value of 0.95. The nodes then have a 5% chance to fail every time a new node is contacted. In reality, hardware failures are way less common and occur only after a prolonged period, so having a high node reliability of 95% is justifiable. With such a configuration we get $Pmin = 70\%$ and $Pmax = 100\%$, despite only allowing 3 recovery attempts. This shows us that the main culprit are node failures, as our failure detection and recovery systems are able to handle communication failures.

5.2 Heartbeat Leader Election

The Heartbeat Leader Election does not contain non-deterministic choices except for the choice of the initiating node. Bully makes non-deterministic choices of which neighbors to contact in what order. In HB Leader Election a single specific neighbor is always contacted based on the required conditions. For this reason the $Pmin$ and $Pmax$ results of verification have the same value, since we have only stochastic behavior, so we will be providing only the $Pmin$ value.

HB Property 1: Correctness of leader election.
Using the same method as with Bully we verified that our HB leader election algorithm is correct when no failures occur.

HB Property 2: Correctness of the failure detection and recovery.

Although it might seem unnecessary to verify the failure detection and recovery system again since the core concept is the same. The specific integration with the leader election need to be verified, especially exiting of recovery mode. Correct exiting from a recovery depends on what type of failure occurred and on the current stage of the algorithm. This is why the recovery system must be verified separately for both leader elections and our result of $Pmin = 100\%$ proves that the system is correct.

HB Property 3.1: Weak leader election success with only communication failures, high reliability and low amounts of recovery attempts.

- $global\_send\_reliab = 0.8$
- $global\_recov\_reliab = 0.9$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$
- $max\_le\_rounds = 2$

A maximum of 2 rounds were permitted and the rest of the network has the same network configuration as in Bully Property 3.1. HB Leader Election has roughly the same odds as Bully with $Pmin = 97\%$.

HB Property 3.2: Weak leader election success with only communication failures, low reliability and low amount of recovery attempts.

- $global\_send\_reliab = 0.8$
- $global\_recov\_reliab = 0.9$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 2$
- $max\_le\_rounds = 2$

Using the same network configuration as in Property 3.1 and a maximum of 2 rounds, we get $Pmin = 34\%$, whereas Bully had $Pmin = 49\%$.

HB Property 3.3: Weak leader election success election with only communication failures, low reliability and high amount of recovery attempts.

- $global\_send\_reliab = 0.4$
- $global\_recov\_reliab = 0.6$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 5$
- $max\_le\_rounds = 2$

When we increase the amount of recovery to 5, it also gives us a significant boost to our odds by bringing it up back to $Pmin = 92\%$.

- $global\_send\_reliab = 0.4$
- $global\_recov\_reliab = 0.6$
- $global\_node\_reliab = 1$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 5$
- $max\_le\_rounds = 3$

If we increase the amount of allowed rounds, our chance actually drops by one 2% to $Pmin = 90\%$. This is most likely caused by more opportunities for failures to happen, as we add roughly 50% more transitions by increasing the amount of rounds from 2 to 3.

HB Property 4.1: Weak leader election success with all types of failures and low node reliability.

- $global\_send\_reliab = 0.7$
- $global\_recov\_reliab = 0.8$

- $global\_node\_reliab = 0.7$
- $global\_reply\_reliab = 1$
- $max\_recovery\_attempts = 5$
- $max\_le\_rounds = 2$

Based on our observation from HB Property 3.3 we decided to keep the amount of rounds to 2. Heartbeat Leader Election performed very poorly with $Pmin =< 1\%$ chance of success. Though we do believe this is caused by the higher number of occurrences of node fail checks compared to Bully, which are responsible for nodes failing. In HB Leader Election every node will trigger fail check at least once, whereas in Bully only the active nodes trigger fail checks and not every node has to be come active. We modified the PRISM model to perform failure checks on every 3rd attempt by introducing a counter a variable $node\_fail\_counter$, that triggers node fail checks when it reached the value 3. With this modification we get a success chance of $Pmin = 5\%$

HB Property 4.2: Weak leader election success with all types of failures and high node reliability.

- $global\_send\_reliab = 0.7$
- $global\_recov\_reliab = 0.8$
- $global\_node\_reliab = 0.7$
- $global\_reply\_reliab = 0.95$
- $max\_recovery\_attempts = 5$
- $max\_le\_rounds = 2$

When we increase the node reliability to 95% we get a result of $Pmin = 8\%$. The modification from Property 4.1 increase it to $Pmin = 50\%$. which is still way lower than Bully. We have abstained from verifying strong success of the leader election based on the results above, since HB performs worse than Bully and with the more strict definition of leader election success the results would be even lower.

Chapter 6

Conclusion

We have modeled two leader election algorithms: a modified version of an existing leader election algorithm called Bully and a second leader election called heartbeat leader election of our own design. A network was modeled that can face both node failures and communication failures. The node failures are permanent, while the communication failures are transient.

Both leader election algorithms were strengthened with a simple failure detection and recovery mechanics, that allows it to function better in such a network. With model checking we have computed the minimum and maximum chance of the leader election being performed successfully under various conditions. The success of leader election can be defined in two ways: weak leader election success and strong leader election success. In the weak definition it is counted as a success if we could not reach a non-failed node, whereas in the strong definition it is counted as a failure.

Using model checking we verified that our leader election algorithms work correctly under full reliability. Then we verified that our failure detection and recovery systems are also correct. Lastly, we verified the leader election algorithms under various network reliability conditions.

Both leader election algorithms are able to perform well in a high reliable network with only small chances of failure. If the reliability of the network goes down below 50%, then we are able to compensate by increasing the recovery attempts, in order to increase the success of the leader election back to an acceptable level. Heartbeat Leader Election performed worse than Bully, especially when node failures are permitted. Though it was possible to improve this by lowering the amount of fail checks being triggered in the system to a amount similar to Bully, which is an unfortunate side consequence of the implemented model. The stricter definition of a strong success has lower odds of success as expected. The results show us that the modified Bully election performs better than the Heartbeat Leader Election of our own design.

We have noticed that allowing more recovery attempts can significantly boost the chance of success when facing communication failures. Node failures are the main reason for leader election to fail and cannot be solved by software additions like this. The selection of the starting node and the topology of the network are also a major factor. If the node that should win the election is starting the leader election, the chance of success is much higher. Moreover if an edge node that is isolated, i.e. has a single communication link through a single neighbor, starts the election, the chances of success are also lower.

We proved that, when facing network unreliability we can improve the effectiveness of the deployed leader election algorithms by adding a simple failure detection and recovery system.

Bibliography

[BBD+06] Simon Bäumler, Michael Balser, Andriy Dunets, Wolfgang Reif, and Jonathan Schmitt. Verification of medical guidelines by model checking – a case study. In *Proceedings of the 13th International Conference on Model Checking Software*, SPIN'06, page 219–233, Berlin, Heidelberg, 2006. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CLNW92] Y.-C. Chow, K.C.K. Luo, and R. Newman-Wolfe. An optimal distributed algorithm for failure-driven leader election in bounded-degree networks. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, pages 136–141, 1992.

[Dil70] David Dill. The murphi verification system. pages 390–393, 01 1970.

[Erc13] Kayhan Erciyes. *Distributed Graph Algorithms for Computer Networks*. Springer Publishing Company, Incorporated, 2013.

[Fix08] Limor Fix. *Fifteen Years of Formal Property Verification in Intel*, page 139–144. Springer-Verlag, Berlin, Heidelberg, 2008.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, jan 1987.

[GCLLR13] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. Fault-tolerant leader election in mobile dynamic distributed systems. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 78–87, 2013.

[GM82] Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.

[HJK+22] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *International Journal on Software Tools for Technology Transfer*, 24:1–22, 08 2022.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, may 1997.

[ISWW09] Rebecca Ingram, Patrick Shields, Jennifer E. Walter, and Jennifer L. Welch. An asynchronous leader election algorithm for dynamic networks. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[JY06] Yi Jiang and Jinyuan You. A low latency chord routing algorithm for dht. In *2006 First International Symposium on Pervasive Computing and Applications*, pages 825–830, 2006.

[KKKK95] Tai Woo Kim, Eui Hong Kim, Joong Kwon Kim, and Tai Yun Kim. A leader election algorithm in a distributed computing system. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 481–485, 1995.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, page 52–66, Berlin, Heidelberg, 2002. Springer-Verlag.

[KNP08] M. Kwiatkowska, G. Norman, and D. Parker. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):14–21, 2008.

[KY76] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[LL77] Gerard Le Lann. Distributed systems - towards a formal approach. pages 155–160, 01 1977.

[San06] Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, USA, 2006.

Appendix A

A.1 Bully Leader Election

Storm 1.7.0 was used. Please note that except for properties 1 and 2, it can take up to several hours to perform the verification when a non-deterministic random starting node is chosen. The longest duration we have observed was 4.5 hours, though it was done on a Macbook Pro with a Apple M1 Pro chip and 16GB of RAM. It is also possible to verify each starting node individually by commenting out the individual command on lines 540-546. For all Bully verification use the file: $final\_network\_7\_nodes\_bully.pm$

- To verify Bully Property 1 setup the network configuration 1 and use the verification formula:
  "$Pmin = ?$ [F"$le\_success\_1$"]".

- To verify Bully Property 2 setup the network configuration 2 and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_1$"]".

- To verify Bully Property 3.1 to 3.3 setup the network configuration for each property as required and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_3$"]".

- To verify Bully Property 4.1 to 4.31 setup thenetwork configuration for each property as required and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_1$"]".

- To verify Bully Property 5.1 setup the network configuration 5.1 and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_4$"]".

- To verify Property 5.2 setup the network configuration 5.2 and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_4$"]".

A.2 Heartbeat Leader Election

For all Bully verification use the file: $final\_network\_7\_nodes\_hble.pm$. The same remarks apply for HB Leader Election regarding the workaround for verifying specific starting nodes.

- To verify HB Property 1 setup the network configuration 1 and use the verification formula:
  "$Pmin = ?$ [F"$le\_success\_1$"]".
- To verify HB Property 2 setup the network configuration 2 and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_1$"]".
- To verify HB Property 3.1 to 3.3 setup the network configuration for each property as required and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_3$"]". For the modification refer to lines 614-619 in the file.
- To verify HB Property 4.1 to 4.2 setup the network configuration for each property as required and use the verification formula:
  "$Pmin = ?$ [F "$le\_success\_final\_current\_3$"]".