University of Camerino

School of Sciences and Technologies

Master degree in Computer Science (LM-18)

Leader Election in a mesh network with Sibilla

Distributed Systems

Author
Francesco Moschella

Matricula 122435

Advisor
Prof. Loreti Michele

A.Y. 2022/2023

Contents

1. Introduction

This document describes the process followed to complete the Distributed Systems project.

In section 1.1, the description of the problem assessed is reported, and in sections 1.2 and 1.3, the problem-related theory and the used framework are introduced.

In Chapter 2, the main focus will be on the project itself and its development, starting from the development process (Section 2.1), followed by some comments and discussion about the code (Sections 2.2 and 2.3) and the test results (Section 2.4).

Finally, in the last chapter (Chapter 3), will be reported some problems (Section 3.1) that affected the workflow and future works (Section 3.2) that might improve the project.

1.1 Project Description

Nowadays, distributed systems are quite common. Users commonly perceive them as a single entity, just like interacting with a single local machine or server.

These systems might be composed of several computing units, sensors and actuators. In the environments created by such technologies, some features are commonly mistaken for granted [3]. Some clear examples are network reliability, the assumption of having a static network topology or even the homogeneity of the nodes belonging to the network.

In some cases, a high degree of distribution also poses a problem in ordinary tasks, such as retrieving execution logs or managing the network or its components.

To assess such problems, a family of algorithms have been created, the *Leader Election Algorithms*, devoted to finding and electing a leader among a group of participants in a distributed system or a network. The leader will then be able to control the network or distributed system and make decisions that influence other nodes.

Another major problem in distributed systems is software behaviour and functionalities testing. This problem derives, for example, from the observability of input and output in the system [4]. Also, it can be expensive to develop a distributed system just for testing software or the behaviour of an algorithm.

It is possible to simulate algorithms and behaviours using specific software or frameworks. In this project, the *Sibilla* framework was used to test the functionality of the leader election algorithm.

1.2 Leader Election Algorithm

In distributed systems, it might be necessary that for certain time slots, a single node has to manage the system or make some specific decision.

Over the years, various approaches have been presented in the literature, [2, 5, 6] are just some examples of the currently available solutions.

In this project, a Leader Election Algorithm will be modelled. The algorithm finds a leader among communicating nodes that participate in a network. To simplify the development, the algorithm is based on the following assumptions:

- The network communication is Bidirectional
- When the leader election algorithm starts, each node has a unique ID (the ID might be generated before or hardcoded)
- The input network topology will not change after the system has initialised the leader election. Therefore, connections between nodes will not change, and nodes will not go offline

Using these assumptions is possible to reach a state when only one leader is present in the network in a finite amount of steps, even if the network is not reliable and there is a probability that messages are lost.

1.3 Sibilla

Sibilla is a Java framework designed to support the analysis of Collective Adaptive Systems [1]. It is a container where different tools supporting specifications and analysis of concurrent and distributed large scaled systems can be integrated. The currently available specification languages defined in the documentation are:

- Language of Interactive Objects (LIO)
- Language of Population Model (LPM)
- Simple Language for Agent Modelling (SLAM)

During the project development, *Language Population Models* have been used.

2. Project Development

This is the central chapter of the report. Here is discussed the development of the project[1], starting with a part dedicated to the code that was developed and the decisions taken (2.1), passing by two sections where some code snippets are shown and discussed (2.2, 2.3) and finally reaching the testing (2.4) where the results are examined.

2.1 Development Process

The project's initial idea was to represent a system where Nodes arranged in a network with a ring topology had to find a Leader.

To develop this project, some assumptions have been considered. First of all, each node is characterised by an ID. The one with the highest ID becomes the Leader. Then, it was assumed that the communication between nodes is bidirectional. Initially, it was assumed that a Node with ID "i" was always followed by a Node with ID "i+1", and preceded by a node with ID "i-1", also since the topology was a ring, the last node was followed by the first one and vice versa, the first node was preceded by the last one. The initial sample version of the code was fully realised using the population models specification language.

In the initial version, there were only two *Species*, the Node (N) and the Follower (F). Having these species, some initial code and testing have been performed.

After this initial implementation, the system seemed too simple. Also, at the end of the simulation, there was no effective leader, but just a node and some followers of that node. The ring topology was a limitation of the algorithm.

After the initial tests, to get acquainted with the language, two more *Species*, the Leader (L) and the Connection between two nodes (C), have been added.

Having these two new species, the code was fixed and completed.

Finally, the possibility to generate the communication graph and provide it at run-time was an interesting feature. A Python script that modifies the LPM code was added. The script adds the system with the generated graph to the LPM file and runs the simulation using the Python user interface provided by Sibilla.

2.2 LPM Code

The code of the Population Model can be split into three main parts:

1. agent definition

[1]The LPM code with its related files are available at https://osf.io/aup82/?view_only=022dd0b7c2ca4e67b3a2a2233f4542d8

2. rules definition
3. system definition

In the first part, there is just some species initialisation and the definition of three parameters: the number of nodes, the rate at which messages are exchanged in the network and the rate at which communication indexes are inverted. The reason for the necessity of the last rate will be discussed in the problems section of the final chapter (Section 3.1).

Following this part, we find the rules definition. Here, the agent's behavioural rules are defined. Assuming that there is a direct connection "C" between the two parts, we have at six cases:

- Node communicates with a Node
- Node communicates with a Follower
- Node communicates with a Leader
- Follower communicates with a Leader
- Follower communicates with a Follower
- Leader communicates with a Leader

Defining "N" the Node, "F" the Follower and "L" the Leader, we can synthesise the rules as follows:

- $N|N \xrightarrow{p} L|F$
- $N|F \xrightarrow{p} L|F$
- $N|F \xrightarrow{p} F|F$
- $N|L \xrightarrow{p} L|F$
- $F|L \xrightarrow{p} F|F$
- $F|L \xrightarrow{p} F|L$
- $F|F \xrightarrow{p} F|F$
- $L|L \xrightarrow{p} L|F$

From the initial six cases, eight rules are derived. Actually, more rules might have been listed. For example, the first one ($N|N \xrightarrow{p} L|F$) could also be mentioned as $N|N \xrightarrow{p} F|L$, and the same logic can be applied to all the other rules.

In some cases, we can also notice that even if the preconditions are the same, the outcome is not. This depends on the piece of information carried by the agents, for example, the second ($N|F \xrightarrow{p} L|F$) and third ($N|F \xrightarrow{p} F|F$) rules, here we have that from the same precondition (a Node communicates to a Follower), is possible to reach two different outcomes, the first one being the generation of a Leader and its Follower. The other case is the creation of two Followers. The two rules assume that in the first case, the node ID is higher than the one carried by the Follower. On the contrary, in the

second case, the Follower transmits an ID higher than the one of the Node. Therefore in the first case, the Node can start pretending to be a Leader and the value in the Follower is updated. In the second case, the Node has to become a Follower and starts transmitting the value received by the other Follower.

2.3 Python Code

To simplify the development and automatise the testing of the LPM code, a simple Python script is provided. The script uses some parameters received via the command line, modifies the LPM file accordingly, and configures and runs the simulation.

The Python script performs four major operations:

- Reads the simulation topology from a LaTeX file
- Applies the specified topology to a clone of the LPM file
- Generates the measures to take into account during the simulation
- Runs the simulation

For simplicity and to make the algorithm adaptable to more use cases, no specific topology was hard-coded in either: the LPM file or the Python script. Instead, a more scalable approach was used by using Tikz and LaTeX. Indeed, one of the parameters required by the script is a Tex file containing a Tikz figure in the form:

```
\begin{tikzpicture}[<options>]
    \node (<node_numeric_ID>) [<node_type>] {node_label};
    ...
    \node (<node_numeric_ID>) [<node_type>] {node_label};

    \path [<options>]
        (<node_numeric_ID>) edge [<edge_type>] {edge_label} (<other_node_numeric_ID>)
        ...
        (<node_numeric_ID>) edge [<edge_type>] {edge_label} (<other_node_numeric_ID>);
\end{tikzpicture}
```

The code only uses the "numeric_ID" for the nodes and the edges.

2.4 Test Results

Two LaTeX files are provided as proof of concept, one representing an ordered ring and the other representing a mesh topology.

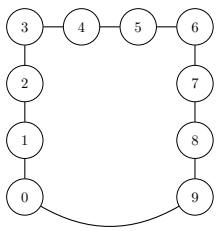

Figure 2.1: Ring Topology

These two graphs, in the form of LaTeX Tikz pictures, have been passed to the python software that extracted for each of them nodes and edges, and then using those, generated an LPM system and ran the simulation.

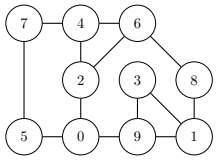

Figure 2.2: Mesh Topology

Ordered Ring Topology In this paragraph, are presented the main results derived from the testing on the ring topology shown in Figure 2.1.

For simplicity and reproducibility of the results, the parameters used in the simulation are:

- Seed: 8797063762540935168
- Amount of replicas: 200
- Deadline: 200
- Delta: 1

While the code of the ring is presented in Figure 2.3

```
\begin{tikzpicture}[node distance = 1.5cm,
sibling distance=10mm, on grid, auto,
every loop/.style={stealth-}]
\node (0) [state] at (0,0) {0};
\node (1) [state, above of=0] {1};
\node (2) [state, above of=1] {2};
\node (3) [state, above of=2] {3};
\node (4) [state, right of=3] {4};
\node (5) [state, right of=4] {5};
\node (6) [state, right of=5] {6};
\node (7) [state, below of=6] {7};
\node (8) [state, below of=7] {8};
\node (9) [state, below of=8] {9};

\path[thick]
    (0) edge [] node {} (1)
    (1) edge [] node {} (2)
    (2) edge [] node {} (3)
    (3) edge [] node {} (4)
    (4) edge [] node {} (5)
    (5) edge [] node {} (6)
    (6) edge [] node {} (7)
    (7) edge [] node {} (8)
    (8) edge [] node {} (9)
    (9) edge [bend left] node {} (0);
\end{tikzpicture}
```

Figure 2.3: Ring Topology Tikz LaTeX code

After the run, results in CSV and corresponding graphs got collected for all the meaningful measures. Following in this document are presented only the most important graphs.

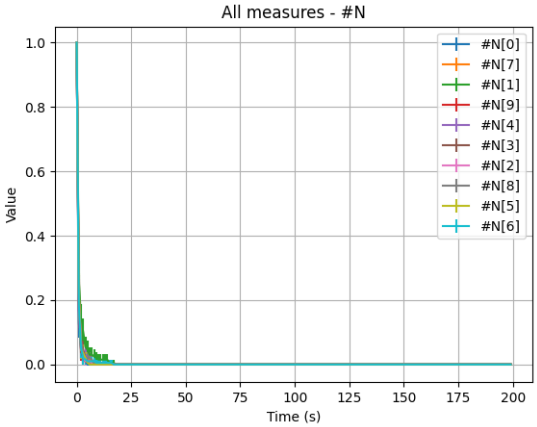

Figure 2.4: Nodes evolution in the system

Figure 2.4 shows the evolution of the Nodes in the system. As is possible to notice, with ten nodes, the system stabilises quite fast. At 25 timesteps, there were already zero species of Nodes, thus meaning that all have changed to either Leader or Follower.

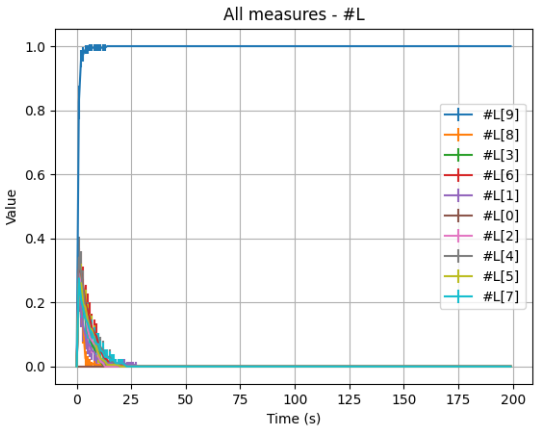

Figure 2.5: Leader evolution in the system

Figure 2.5 presents the evolution of the leaders in the system. As is possible to notice, with ten nodes in the system, the actual Leader is found in little more than 25 timesteps. After that, only one Leader remains, thus meaning that all the other nodes have changed to followers of the specified Leader (N[9] in the Figure).

Although the graph in Figure 2.6 is more confusing than the others, two distinct and symmetrical behaviours are present. Part of the lines moves towards 1, and the other group goes to 0. Again the system stabilises in a little more than 25 timesteps. The two groups are the number of followers for each node. Among all the possible combinations of Follower with index "i" of Leader with index "j", the group that stabilises as present (1 in the graph) are the ones for which the "j" values correspond to the actual Leader (9 in the Figure).

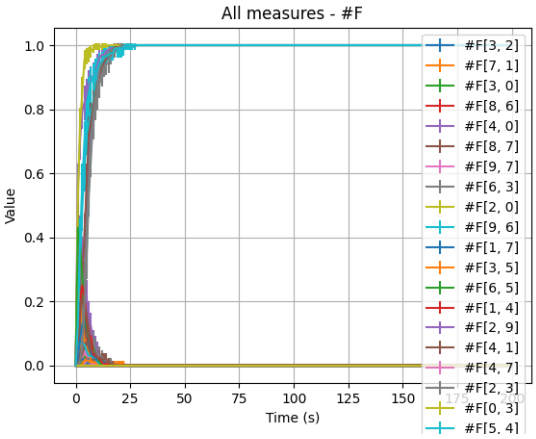

Figure 2.6: Follower evolution in the system

Since the results are so hard to notice with this high amount of timesteps, the same test was repeated, using the same parameters, but for a lower amount of timesteps: 30.

Following are present the graphs regarding the new test phase.

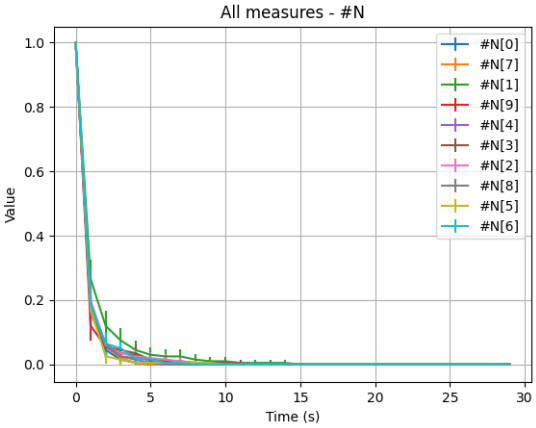

Figure 2.7: Nodes evolution in the system

Using the previous graphs (Figures 2.7, 2.8, 2.9), is possible to have a closer look on the system behaviour during the initial 30 timesteps.

While Nodes and Leader graphs should be clear, the chart provided for the Followers might confuse the reader. The main problem resides in the colour chosen by Matplotlib. Indeed the same colour or its shadows have been used to represent different lines. For example, the purple line increasing towards 1 in the graph may be mistaken for the one presented as "#F[4, 0]" in the legend instead of "#F[2, 9]".

Mesh Topology Tests In this paragraph, the most significant results, derived from the testing on the mesh topology shown in Figure 2.2, are presented.

For simplicity and reproducibility of the results, the parameters used in the simulation are:

- Seed: 5042599843285415936

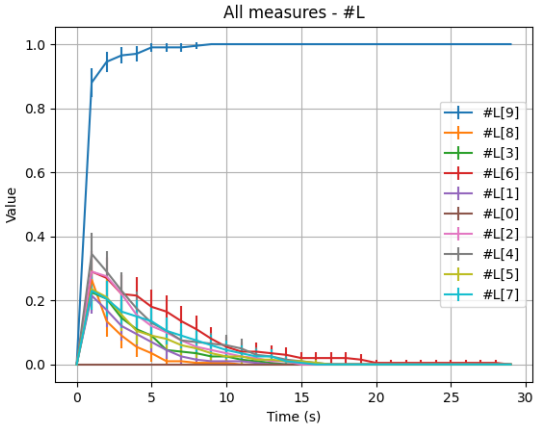

Figure 2.8: Leaders evolution in the system

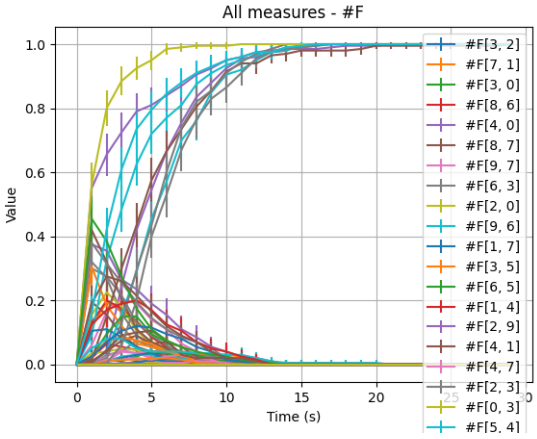

Figure 2.9: Follower evolution in the system

- Amount of replicas: 200
- Deadline: 200
- Delta: 1

While the code of the mesh is presented in Figure 2.10

```
\begin{tikzpicture}[node distance = 1.5cm,
    sibling distance=10mm, on grid, auto,
    every loop/.style={stealth-}]
\node (0) [state] at (0,0) {0};
\node (5) [state, left of=0] {5};
\node (2) [state, above of=0] {2};
\node (9) [state, right of=0] {9};
\node (1) [state, right of=9] {1};
\node (3) [state, above of=9] {3};
\node (8) [state, above of=1] {8};
\node (6) [state, above of=3] {6};
\node (4) [state, above of=2] {4};
\node (7) [state, left of=4] {7};

\path[thick]
    (0) edge [] node {} (2)
    (1) edge [] node {} (3)
    (1) edge [] node {} (8)
    (4) edge [] node {} (7)
    (1) edge [] node {} (9)
    (5) edge [] node {} (0)
    (2) edge [] node {} (6)
    (4) edge [] node {} (6)
    (3) edge [] node {} (9)
    (2) edge [] node {} (4)
    (7) edge [] node {} (5)
    (6) edge [] node {} (8)
    (9) edge [] node {} (0);
\end{tikzpicture}
```

Figure 2.10: Mesh Topology Tikz LaTeX code

After the run, results in CSV and corresponding graphs got collected for all the meaningful measures. Following in this document are presented only the most important graphs.

Figure 2.11 shows the evolution of the Nodes in the system. As is possible to notice, with ten nodes, the system stabilises quite fast. At 25 timesteps, there were already zero species of Nodes, thus meaning that all have changed to either Leader or Follower.

Figure 2.12 presents the evolution of the leaders in the system. As is possible to notice, with ten nodes in the system, also the actual Leader is found before the 25^{th} timestep, thus meaning that all the other nodes have changed to followers of the specified Leader (N[9] in the Figure).

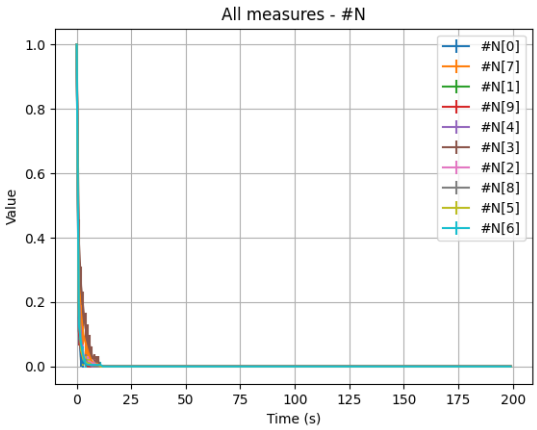

Figure 2.11: Nodes evolution in the system

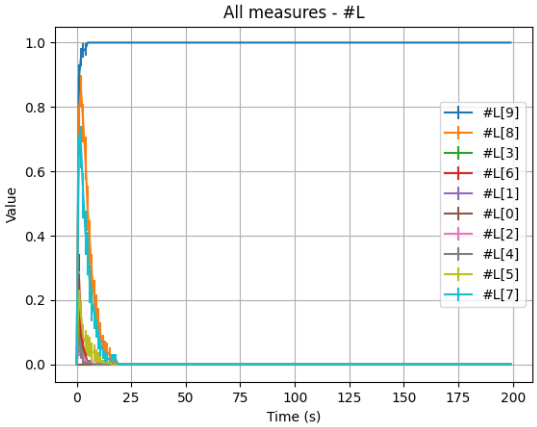

Figure 2.12: Leader evolution in the system

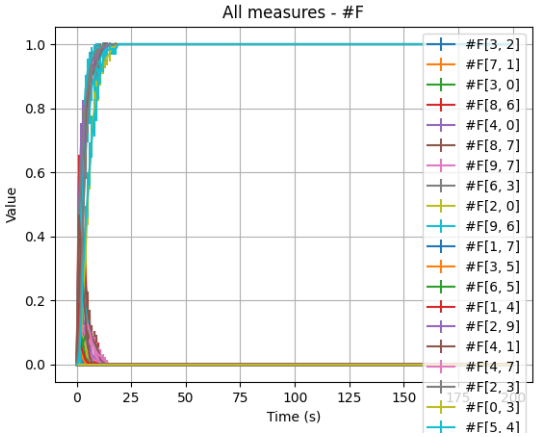

Figure 2.13: Follower evolution in the system

As previously mentioned, the graph in Figure 2.13 appears to be more confusing compared to the others, since the plot presents two distinct and symmetrical behaviours. Part of the lines moves towards 1, and the other group goes to 0. Again the system stabilises in a little less than 25 timesteps. The two groups are the number of followers for each node. Among all the possible combinations of Follower with index "i" of Leader with index "j", the group that stabilises as present (1 in the graph) are the ones for which the "j" values correspond to the actual Leader (9 in the Figure).

Since the results are so hard to notice with this high amount of timesteps, the same test was repeated, using the same parameters, but for a lower amount of timesteps, 25.

Following are present the graphs regarding the new test phase.

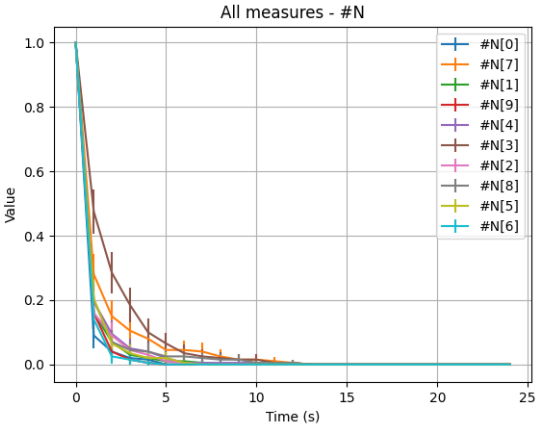

Figure 2.14: Nodes evolution in the system

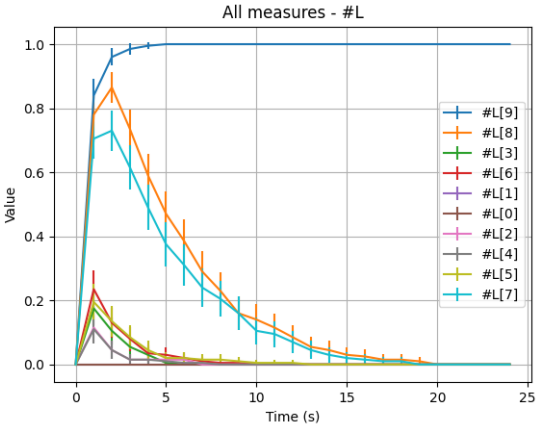

Figure 2.15: Leaders evolution in the system

Using the previous graphs (Figures 2.14, 2.15, 2.16), is possible to have a closer look on the system behaviour during the initial 25 timesteps.

Again, while graphs related to Nodes and Leader should be clear, the chart provided for the Followers might be confusing. Because of colours chosen by Matplotlib. Indeed the same colour or its shadows have been used to represent different lines. For example, the purple line increasing towards 1 in the graph may be mistaken for the one presented as "#F[4, 0]" in the legend instead of "#F[2, 9]".

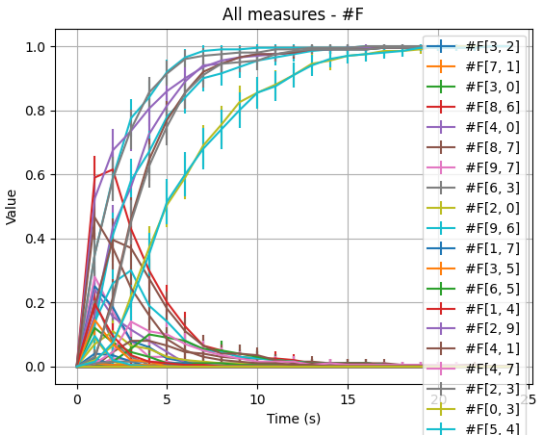

Figure 2.16: Follower evolution in the system

3. Conclusion

During the development of this project, was learnt how to specify population models and some of the features provided by Sibilla.

It was a pleasant learning opportunity and a fun experience. Unfortunately, some limitations in the specification language and the Python interface have been found, which have been managed by furtherly studying and getting to know the background of the framework.

In the following sections, some issues faced during the development (Section 3.1) will be discussed and some future works (Section 3.2) that may increase the functionalities currently available in my script and model will be presented.

3.1 Problems

Initially, the only information available was Sibilla getting started and the lecture material.

Some functionalities were unknown, and some others were mistaken for granted. This was a limitation and slowed down the initial development stages.

The Sibilla shell was initially counterintuitive. Some commonly present features in command line interfaces like the help section and the possibility to use the Tab key to autocomplete the commands or the paths were not provided, hence negatively impacting the overall user experience.

After learning the shell commands and functionalities, the Python User Interface was straightforward and intuitive. The principal problems encountered are the lack of typed function inputs or return values. Even if Python is a dynamically typed language, as of Python 3.5 is possible to set types to function parameters and return values. This is quite useful, especially for the IDEs, since the typing allows the IDE Code Completion engine to perform better. Also, the code documentation permits understanding the behaviour of a called method and correctly using it. The Python library code and typing information were present in the Getting Started guide. But not in the library source code, therefore, the IDE had problems recognising output types and marked valid function calls like "plot(...)" with a warning.

During the development, there was a problem with the plotting functions in the Python script. After the simulation completion, Python could not plot the graphs because of an exception in the Sibilla Python library. Since the error message was not meaningful, it was necessary to overcome this problem by asking for help. It has been discovered that the "high" (more than ten) amount of parameters might cause this error.

After solving the initial problems and being able to extract some graphs using the Python Matplotlib library, an error in the LPM code was discovered. The node con-

nection is passed as part of the system specification in the form "C[i, j]", with "i" and "j" identifiers of two connected nodes in the network. After the test phase for the mesh network (Figure 2.2), Nodes 6 and 3, at the end of the simulation, instead of being Followers of the Leader node 9, were still nodes in a high amount of cases (Figure 3.1).

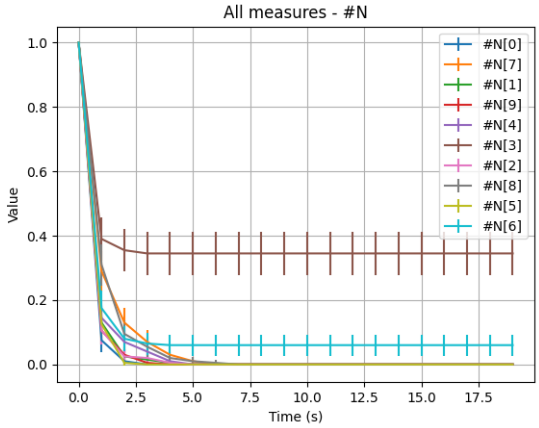

Figure 3.1: Initial simulation result for the amount of Nodes species

I figured out that the error was caused by the "when" clause used in the rules to correctly select the specific rules for changing the species of an entity. Indeed in most cases, one index had to be lower than the other, not allowing configurations where the connection might be in the form "C[9, 3]". To solve this fallacy, was add a rule. The latter, with a probability of 0.3, inverts the indexes of a connection.

When testing the mesh, cases with two elected Leaders were discovered, specifically nodes 8 and 9 (Figure 3.2).

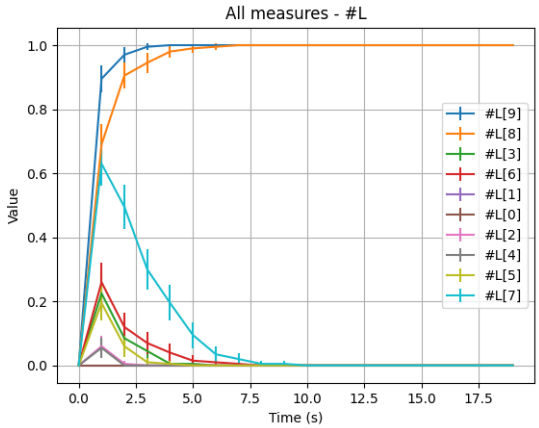

Figure 3.2: Initial simulation result for the amount of Leaders species

I went through the LPM code and found that some rules did not consider the equality between the indexes used by leaders and followers, thus not updating the species and resulting in the Followers continuously changing their followed leader. To solve this problem, $<$ or a $>$ were respectively changed to $\leq$ or $\geq$.

3.2 Future Works

Even if the model and script seem to work, there are some problems. Firstly, the graphs can use only the LaTeX Tikz "automata" library. Therefore, many other kinds of graphs equivalent to the one presented can be realised, but are not readable by the Python script since they might not use the same format. I'm also considering lowering the assumptions, such as the bidirectional communication and the absence of node downtimes. Specifically, it would be interesting to model a system where the network topology might change during the simulation. To achieve this, a rule in the form $C[i,j] \xrightarrow{p} C[i,k]$ can be added to the LPM file. Actually, this rule might add multiple connections between two nodes, but this problem should not introduce bugs or errors, except for some communication between nodes that happen more than one time at each timestep. Also, it would be interesting to allow nodes to go offline with some probability, thus changing the network topology and, with a certain probability, the leader. It should be noted that to obtain that behaviour, multiple rules should be added and a simple rule such as $N[i] \xrightarrow{p} O[i]$ would not be sufficient. For instance, we would need at least one rule to turn offline each specie, a rule to turn the offline species back online, and some other rules to detect the absence of the leader and start a new election. Also, studying the different behaviour when the communication between nodes cannot be assumed bidirectional might be intriguing. This model assumes that all nodes want to participate in the leader election, which might not always be correct. Therefore it could be fascinating to specify an LPM where only some participants are interested in becoming leaders, this can be achieved by considering the not interested nodes as offline. Some of these functionalities could also be mimicked using the current LPM file. For example, by providing as initial parameters some followers is possible to represent their absence of interest in participating in the leader election. As a side note, the provided files[1] contain a work-in-progress folder with an extended model incorporating a set of rules that change the neighbours of each node at runtime, representing some initial steps towards the possibility of nodes going offline. Finally, the current and extended models might need some fixes and tweaks to behave better.

[1]The LPM code with its related files are available at https://osf.io/aup82/?view_only=022dd0b7c2ca4e67b3a2a2233f4542d8

Bibliography

[1] Nicola Del Giudice et al. "Sibilla: A tool for reasoning about collective systems". In: *Coordination Models and Languages: 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*. Springer. 2022, pp. 92–98.

[2] Rebecca Ingram et al. "An asynchronous leader election algorithm for dynamic networks". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–12.

[3] Arnon Rotem-Gal-Oz. "Fallacies of distributed computing explained". In: *URL http://www.rgoarchitects.com/Files/fallacies.pdf* 20 (2006).

[4] Werner Schütz. "Fundamental issues in testing distributed real-time systems". In: *Real-Time Systems* 7.2 (1994), pp. 129–157.

[5] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. "Design and analysis of a leader election algorithm for mobile ad hoc networks". In: *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.* IEEE. 2004, pp. 350–360.

[6] Sudhani Verma, Divakar Yadav, and Girish Chandra. "Leader election algorithm in fault tolerant distributed system". In: *Proceedings of Second Doctoral Symposium on Computational Intelligence: DoSCI 2021*. Springer. 2022, pp. 471–480.