Self-Organizing and Self-Stabilizing Sensor Network

ARD

Students:
Amir Menczel
Radmila Fishel
Self-Organizing and Self-Stabilizing Sensor Network

ARD

Students:
Amir Menczel
Radmila Fishel

Advisors:
Prof. Shlomi Dolev
Dr. Armin Shmilovici
Table of Contents

1. Introduction ................................................................................................................. 5
   1.1 Vision .................................................................................................................... 5
1.2 The Problem Domain ................................................................................................. 5
   1.2.1 Overview .......................................................................................................... 5-6
   1.2.2 Legend ................................................................................................................. 6
   1.2.3 Main Problem ..................................................................................................... 7-8
1.3 Stakeholders ............................................................................................................... 8
   1.3.1 Customers .......................................................................................................... 8
   1.3.2 Users ................................................................................................................... 8
1.4 Software Context ....................................................................................................... 8
   1.4.1 Cluster Partitioning .......................................................................................... 9
   1.4.2 Hierarchy construction ..................................................................................... 9-11
   1.4.3 Faults ................................................................................................................ 11
   1.4.4 Adding new sensors ........................................................................................ 12
   1.4.5 Network stabilization ...................................................................................... 13
1.5 System Interfaces ..................................................................................................... 13
   1.5.1 Hardware Interfaces ......................................................................................... 13
   1.5.2 Software Interfaces .......................................................................................... 13-14
      1.5.2.1 Simulation Environment .......................................................................... 14-15
   1.5.3 Events ................................................................................................................. 15
2. Functional Requirements ............................................................................................ 15
3. Non-functional requirements ....................................................................................... 16
   3.1 Performance constraints ....................................................................................... 16
      3.1.1 Usability ......................................................................................................... 16
   3.2 Platform Constraints ............................................................................................. 16
   3.3 Special restrictions & limitations .......................................................................... 16
4. Usage Scenarios ......................................................................................................... 17
   4.1 User Profiles — The Actors ............................................................................... 17
      4.1.1 User ................................................................................................................. 17
      4.1.2 SHAWN ......................................................................................................... 17
   4.2 Use-cases .............................................................................................................. 18-25
   4.3 Special usage considerations ................................................................................. 26
5. Appendices

5.1 Appendix 1 – Glossary

5.2 Appendix 2 – SHAWN

5.2.1 Main Concept

5.2.2 Implementation

5.2.3 Design

5.2.4 Architecture

5.2.4.1 Models

5.2.4.1.1 Communication model

5.2.4.1.2 Edge Model

5.2.4.1.3 Transmission Model

5.2.4.2 Sequencer

5.2.4.2.1 Simulation Tasks

5.2.4.2.2 Simulation Controller

5.2.4.2.3 Event Scheduler
1 Introduction

1.1 Vision

The FRONTS project is a joint effort of eleven academic and research institutes in foundational algorithmic research in Europe. The aim of the project is to provide a unifying scientific framework and a coherent set of design rules, for global systems resulting from the integration of autonomous interacting entities, dynamic multi-agent environments and ad-hoc mobile networks. Ben Gurion University is one of the academic institutes that participate in the FRONTS project. This project is a research on self-stabilizing and self-organizing algorithms for sensor networks.

The main goal of our project is to implement distributed algorithms for sensor networks in a way it will be resistant to changes and faults. The simulations will be executed using the SHAWN simulation tool.

The first step is to construct a solid foundation for sensor networks and simulate different network topologies. The network will be partitioned randomly and the entire network must be able to self-stabilize and self-organize fast.

1.2 The Problem Domain

1.2.1 Overview

Sensor networks are used to sample the environment for sensory information (E.g. temperature) and propagate this data to a central point.

Wireless sensors have a limited energy that keeps them working. Usually, when sensors are designed to work outdoors their only power supply is solar energy. For this reason, efficient algorithms must be developed to save as much energy as possible. Efficient algorithm is measured by successfully reducing the amount of messages sent through the network. Those messages are used for communication purposes between the sensors. The complexity of all algorithms that dealt with sensors networks so far was not bounded and depended on the network's diameter. Furthermore, only few algorithms deal with topology changes and those who do, have complexity that is not bounded.
We would like to develop a new asynchronous, randomized, self-stabilizing and self-organizing distributed algorithm for cluster definition in communicating graphs of bounded degree processors. This algorithm will converge within $O(\log^2 n)$ expected number of rounds.

### 1.2.2 Legend

In this document we deal with sensor networks. Simple sensors (i.e. processors), as in Table 1 (a), receive messages, make some computation and can send a new message through their outgoing links.

Simple sensors locate a leader sensor (Table 1, b) in their neighborhood and join its cluster. These leader sensors have a pre-defined purpose in addition to the roles of simple sensors. They create a new layer in the hierarchy and communicate with other leader sensor in this layer. This way, simple sensors can send their messages to remote sensors more efficiently.

The link (Table 1, c) represents a communication channel between two processors. $X$ (Table 1, d) represents the radius of a cluster. For example, assume $X=2$. This parameter implies that the farthest sensor in a cluster is located at most two units of distance apart from the cluster's leader.

<table>
<thead>
<tr>
<th>A</th>
<th>processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>leader</td>
</tr>
<tr>
<td>C</td>
<td>link</td>
</tr>
<tr>
<td>D</td>
<td>radius of a cluster</td>
</tr>
</tbody>
</table>

**Table 1**
1.2.3 Main Problem

A well-known problem that is dealt in synchronous / asynchronous networks is leader election. Many algorithms were suggested as solutions to this problem. Our specific domain deals with distributed networks with arbitrary form and capacity. Distributed systems must allow the processes to share information with each other. The exchanging of information is done using different types of messages that are sent in the network. Efficiency is crucial for running times and energy consumption of the processors in the network. For this purpose, the concept of "Leader Election" was brought up. The chosen leader manages the traffic in the network and also routes messages between the processors. The routing is done faster with the leader than it would without it, because the leader finds shortest paths.

Previous works have tried using the processors' identifiers in order to break symmetry in the leader election algorithm. However, occasionally an unfortunate order of id's may lead to a convergence time which is proportional to the diameter of the graph. This Application will use randomness to break ties in order to overcome such a scenario.

The most familiar types of networks, for which many algorithms for leader election were written, are the ring (figure 1), and sensor network (figure 2).

Our project deals with the type presented in figure 2.

![Figure 1](image)
1.3 Stakeholders

1.3.1 Customers
FRONTS project is established and funded by the EU Union. This project is an integral part of the FRONTS project and in accordance to their supplied demands.

1.3.2 Users
There are two types of users to our project, one type is the researchers. The researchers will be able to simulate different scenarios that can happen to real sensors outdoors and be prepared for serious faults that might occur.

The second type is the operators of the real sensor networks. Once the algorithm will be used later on in real operational sensor networks environments, the network operators will be able to maintain the network easily by using the simulation to predict future malfunctions. They will also be able to see the effect of adding new sensors to an existing network for the purpose of expanding it to a larger area.

1.4 Software Context
The main objective of our software is to simulate different scenarios that can occur in asynchronous sensor networks using efficient distributed algorithms. In this section we will describe the main goal functionalities of the system.
1.4.1 Cluster Partitioning

In the network we deal with, many leaders will need to be elected in order to manage vast amount of messages that will be sent in the network. Every leader will manage a group of sensors. For this to be possible, first, we would like to divide the entire network into smaller clusters. Each cluster will contain a group of sensors and one leader, as can be seen in figure 3. The diameter of a cluster will be set according to the $X$ parameter (diameter = $2X$). Processors that compose the cluster will determine its form, which won’t be uniform.

1.4.2 Hierarchy construction

The network is transformed into hierarchy of layers. This new approach enables us to achieve better running times by reducing the amount of messages that are transmitted in the network. The first layer is based on traditional point-to-point neighboring communication, where communication is between processors that are directly connected by physical communication mean.

First, sensors spontaneously wake up and look for a leader within distance $X$. When no leader is found, they announce themselves as candidates and if possible convert themselves into leaders later on (Figure 4).

Leaders within distance $X$ from each other are eliminated and one leader is left for every $X$-cluster.
Then, other nearby sensors join their cluster (Figure 5).

Finally, the construction of the lower level in the hierarchy is completed (Figure 6).
The participants of a higher level are those that were elected as leaders at the preceding layer. In other words, leaders of leaders are elected and proceed to the next layer. At the end one leader is chosen to be the main leader of the entire network.

1.4.3 Faults

Sensors are powered by solar energy. Therefore their life span is very limited. When a sensor is run out of energy it stops functioning and the other nearby sensors need to adjust to the new state. When a sensor stops functioning, it can be seen as if the links that are directly connected to this sensor are removed from the network (figure 7).

![Figure 7](image)

This kind of fault is referred to as a topology change. The network must be able to recover from a topology change quickly in order to allow the entire network to keep functioning. Sensors that routed their messages through the faulty sensor will need to update their routing table in order to route their messages in a new path that does not include the faulty sensor.

Another type of fault is referred to as "fault containment". This happens when a sensor continues to communicate with its neighbors but the computation it does in every round is wrong. This fault can be caused due to hardware malfunction (e.g. registers are broken down). This fault cannot be fixed.
1.4.4 Adding new sensors

Sensors wake up spontaneously in asynchronous systems. Therefore, a new sensor would like to join the network after it has been stabilized. The stabilized network can be viewed in figure 6. Once a new sensor joins the network, it can join as a regular sensor (Figure 8) or as a leader in case that it is in distance larger than $X$ from the nearest leader (Figure 9).

Figure 8

Figure 9
1.4.5 Network stabilization

The network must be able to self-stabilize and self-organize from any initial state. Figure 10 presents an arbitrary sensor network. As seen below, there are no defined clusters in the network and no hierarchies. The network should be stabilized and transformed to a state that is similar to the one presented in figure 6.

![Figure 10](image)

1.5 System Interfaces

1.5.1 Hardware Interfaces

Not applicable.

1.5.2 Software Interfaces

The application development will be highly relied on the SHAWN simulator tool. Shawn's architecture that is shown in the figure 11 comprises three major parts:

- Models
- Sequencer
- Simulation Environment
The application particularly uses the interfaces supplied by the Simulation Environment. The Models and the Sequencer will be explained in the Appendix.

1.5.2.1 Simulation Environment

The simulation environment is the home of the virtual world in which the simulation objects reside. As shown in figure 11, the simulated Nodes reside in a single World instance. The Nodes themselves serve as a container for so-called Processors.

The self-stabilizing and self-organizing algorithm will be implemented inside the processors. By decoupling the application inside a Processor from the Node, multiple applications can easily be combined in a single simulation run without changing their implementations. For instance, one processor could implement an application specific protocol while another processor gathers statistics data.

The listing below shows Processor’s API. After a processor has been instantiated, its boot() method is invoked. A Processor can transmit messages by a call to send() and whenever a message for the Node is received, it dispatches this message to all its
Processors by calling process_message(). The Processor’s work() method is invoked whenever the Event Scheduler starts a new simulation round.

**Processor API:**
```cpp
void boot( );
bool process_message( MessageHandle& );
void work( );
Node& owner( );
void send( MessageHandle& );
```

A Node offers a number of services to the Processors that ease the implementation of algorithms and simplify the overall simulation task. For example, the Edge Model can be used to identify the neighbors of the current node.

### 1.5.3 Events

The main events of the application are described as uses cases in section 4.2.

### 2 Functional Requirements

This application includes an implementation of a new and unique algorithm that has the properties of self-stabilizing and self-organizing. It will be possible to simulate the algorithm on different topologies of networks. This way we will be able to measure the quality of this new algorithm. Section 4.2 describes the main functionality of the system by describing the use cases.
3 Non-functional requirements

3.1 Performance constraints

As mentioned in the “Vision”, this application is a research about self-stabilizing and self-organizing algorithms in sensor networks. For this reason, there aren’t any non-functional requirements regarding performance issues except of usability.

3.1.1 Usability

1. The application GUI should be user-friendly.
2. The application should be simple to manage for the common user.
3. Not overwhelm the user with redundant messages.

3.2 Platform Constraints

The application will be developed in C++ under Linux operating system, using SHAWN simulator of sensor networks.

The main customer requirement was to use the SHAWN simulator, which is open-source and to keep the development open-sourced. Another customer requirement was to use C++ programming language, the same programming language as SHAWN was developed with.

For this reason we are obliged to develop this application using C++ as well. In order to keep the development open-source, we chose to work under Linux operating system that is also open-source.

After deciding to use Linux, there are a few types of IDE’s we could choose. Linux has many text editors to use, some are more advanced than others’ like Emacs and XEmacs. Still, Eclipse IDE is the most powerful IDE available under LINUX that has support for C++. It offers debugging, refactoring etc.

For conclusion, we will develop the application using the simulator SHAWN using eclipse with C++ plug-in under Linux.

3.3 Special restrictions & limitations

Not applicable.
4 Usage Scenarios

4.1 User Profiles — The Actors

4.1.1 User

The system has one type of user. In practice there are researchers and operators that will use the system. Both of them will use the system in the same way but for different purposes.

4.1.2 SHAWN

The simulation tool (i.e. SHAWN) provides interfaces for real sensors, communication links and messages. In addition SHAWN provides us the capability of manipulating the entire network and presenting it a graphical manner.
4.2 Use-cases

Use Case Diagram

Figure 12
Use Case UC1: View sensor information.

Primary Actor: User

Stakeholders and Interests: User: Wants to view the information about a specific sensor on the network.

Preconditions: The network was already constructed and sensors were initialized.

Post Conditions: The information requested about a specific sensor is presented to the user.

Main Success Scenario:
1. User right clicks a sensor and chooses to view sensor’s information. User also has the option to enter a sensor’s ID in order to view its status information.
2. The sensor’s information is presented in a window.

Extensions (Alternatives):
1a. User enters an ID of a sensor that doesn’t exist.
   1. System presents a relevant error message.

Sequence Diagram:

![Sequence Diagram](view_sensor_information.png)

Figure 13
Use Case UC2: Define network configuration.

Primary Actor: User

Stakeholders and Interests: User: Wants to construct a new network topology.

Preconditions: Application is up and running.

Post Conditions: A new network topology is constructed, ready for use.

Main Success Scenario:

1. User chooses to construct a new network.
2. User selects number of sensors to use.
3. User selects network topology from the given topologies. He also can choose a random topology.
4. User presses done and the new network is presented on screen.

Extensions (Alternatives):

2a. Number of sensors can’t be less than 1. Input must be numeric.

1. System presents a relevant error message.

Sequence Diagram:

![Sequence Diagram](image-url)
Use Case UC3: Save network configuration.

Primary Actor: User

Stakeholders and Interests: User: Wants to save current network configuration for later use.

Preconditions: A network must be constructed first and presented on screen.

Post Conditions: Current network configuration is saved and can be loaded by user choice.

Main Success Scenario:

1. User selects to save the current network.

Extensions (Alternatives):

None.

Sequence Diagram:

Save network configuration

User

:EventHandler

:NetworkHandler

1. saveNetwork(filename)

2. saveNetwork(filename)

3. saveCurrentNetwork(filename)
Use Case UC4: Load network configuration.

Primary Actor: User

Stakeholders and Interests: User: Wants to load a network configuration that was previously saved.

Preconditions: A network must be previously saved as a file and located in the application directory.

Post Conditions: A network was loaded from a file and has the same configuration as the network saved earlier.

Main Success Scenario:
1. User selects a network to load from the application directory.

Extensions (Alternatives):
1a. User selects a file that was not created by the application or is corrupted.
   1. System presents a relevant error message.

Sequence Diagram:

```
Figure 16
```

**Load network configuration**
Use Case UC5: Add a new sensor to the current network used.

Primary Actor: User

Stakeholders and Interests: User: Wants to add a new sensor at an exact coordinate of the network and see the effect of it on the rest of the network.

Preconditions: A network must be constructed / loaded and presented on screen.

Post Conditions: A new sensor is added to the network. The algorithm made the appropriate changes in order to stabilize the network after adding it a new sensor.

Main Success Scenario:
1. User selects a coordinate on the network presented on screen.
2. User right clicks this coordinate and chooses to add a new sensor.

Extensions (Alternatives):
2a. User right clicks on an existing sensor.
   1. The choice of "Add Sensor" is disabled.

Sequence Diagram:

![Sequence Diagram](image)
Use Case UC6: Remove a sensor from the current network presented on screen.

Primary Actor: User

Stakeholders and Interests: User: Wants to remove a sensor from the network and see the effect of it on the rest of the network.

Preconditions: A network must be constructed / loaded and presented on screen.

Post Conditions: A new sensor is removed from network. The algorithm made the appropriate changes in order to stabilize the network after removing the sensor.

Main Success Scenario:
1. User selects to a sensor in the network presented on screen.
2. User right clicks on the sensor and chooses to remove the sensor.

Extensions (Alternatives):
None.

Sequence Diagram:
Use Case UC7: Automatic simulation for a relative long period of time. The application will decide randomly to add / remove a sensor. The network will stabilize after every change.

Primary Actor: User

Stakeholders and Interests: User: Wants the application to make random changes and measure the algorithm performance after every change, for a long period of time while many changes have occurred.

Preconditions: A network must be constructed / loaded and presented on screen. A time for the automatic simulation must be set.

Post Conditions: By the end of the time set for the automatic simulation, the network should be stabilized. A log file will be created in order to document every change made and the time needed for stabilization.

Main Success Scenario:
1. User selects the automatic simulation.
2. User sets the time for the automatic simulation.
3. System starts the automatic simulation.
4. System creates a log file that documents every topology change made and time took for the network to stabilize.

Extensions (Alternatives):
2.a. input is non-positive.
1. System presents a relevant error message.

Sequence Diagram:

Start Automatic Simulation

Figure 19
4.3 Special usage considerations

Not applicable.

5 Appendices

5.1 Appendix 1 – Glossary

**Synchronous** – A telecommunication system in which transmitting and receiving apparatus operate continuously at substantially the same rate, and correction devices are used, if necessary, to maintain them in a fixed time.

**Asynchronous** – In a synchronous system, operations are coordinated under the centralized control of a fixed-rate clock signal or several clocks. An asynchronous digital system, in contrast, has no global clock: instead, it operates under distributed control, with concurrent hardware components communicating and synchronizing on channels.

**Snapshot Algorithm** – The snapshot algorithm is an algorithm used in distributed systems for recording a consistent global state of an asynchronous system. It is also known as Chandy-Lamport Algorithm for the determination of consistent global states.

**Self-Stabilization** – A distributed algorithm is self-stabilizing if starting from an arbitrary state, it is guaranteed to converge to a legitimate state and remain in a legitimate set of states thereafter. A state is legitimate if starting from this state the algorithm satisfies its specification. The property of self-stabilization enables a distributed algorithm to recover from a transient fault regardless of its nature. Moreover, a self-stabilizing algorithm does not have to be initialized as it eventually starts to behave correctly regardless of its initial state.

**Self-Organization** – An algorithm is self-organizing if it converges in sublinear time and reacts "fast" to topology changes. If \( s(n) \) is an upper bound on the convergence time and \( d(n) \) is an upper bound on the convergence time following a topology change, then \( s(n) \in O(n) \) and \( d(n) \in O(s(n)) \).
5.2 Appendix 2 – SHAWN

Shawn is an open-source discrete event simulator that has considerable differences to all other existing simulators. Shawn is very powerful in simulating large scale networks with an abstract point of view. It is, to the best of our knowledge, the first simulator to support generic high-level algorithms as well as distributed protocols on exactly the same underlying networks.

5.2.1 Main Concept

Its central idea is to replace low-level effects with abstract and exchangeable models, such that the simulation can be used for huge networks in reasonable time whilst keeping the focus on the actual research problem.

5.2.2 Implementation

For maximum performance, Shawn is written in C++ and is known to run under Windows and many flavors of Unix/Linux. Released under the BSD license, it is currently in active development and use by different universities and companies to simulate wireless (sensor) networks. It was and continues to be an invaluable tool for many research publications and it has been successfully applied in numerous bachelor and master theses.

5.2.3 Design

The design rationale behind Shawn is to simulate the effects of a phenomenon and not the phenomenon itself by using abstract models and implementations of these models. A user can select the actual granularity and behavior and is hence able to adapt the simulation to his specific needs. E.g., to let message transmissions always succeed within constant time, with probabilistic message loss or, if physical accuracy is really necessary, by simulating the effects of a complete OSI stack.
5.2.4 Architecture

This section includes details about the other parts in SHAWN architecture (i.e. Models and the Sequencer) that were omitted from 1.5.2. SHAWN architecture can be seen in figure 11.

5.2.4.1 Models

Shawn distinguishes between models and their respective implementations. A model is the interface used by Shawn to control the simulation without any knowledge on how a specific implementation may look like. Shawn maintains a repository of model implementations that can be used to compose simulation setups by selecting the desired behaviors.

Three models form the foundation of Shawn:
- Communication Model
- Edge Model
- Transmission Model

5.2.4.1.1 Communication model

Whenever a simulated sensor node in Shawn transmits a message, the potential receivers of this message must be identified by the simulator. Please note that this does not determine the properties of individual transmissions but defines whether two nodes can communicate as a matter of principle. This question is answered by implementations of the Communication Model. The following code presents the C++ interface of the Communication Model. A single method is invoked to determine whether the node b is in reach of the node a.

**Communication Model API**

```cpp
bool can_communicate_uni (Node& a, Node& b);
```
5.2.4.1.2 Edge Model

The Edge Model provides a graph representation of the network. The simulated nodes are the vertices of the graph and an edge between two nodes is added whenever the Communication Model returns true. To assemble this graph representation, the Edge Model repeatedly queries the Communication Model. It is therefore possible to access the direct neighbors of a node, the neighbors of the neighbors, and so on. This is used by Shawn to determine the potential recipients of a message by iterating over the neighbors of the sending node. The following code presents the C++ interface of the EdgeModel.

**Edge Model API:**

```cpp
adjacency_iterator begin_adjacent_nodes (Node &, CommunicationDirection d = bidi);
adjacency_iterator end_adjacent_nodes (Node &);
```

5.2.4.1.3 Transmission Model

Whenever a node transmits a message, the behavior of the transmission channel may be completely different than for any other message transmitted earlier. For instance, cross traffic from other nodes may block the wireless channel or interference may degrade the channel’s quality. To model these transient characteristics inside Shawn, the Transmission Model determines the properties of an individual message transmission. It can arbitrarily delay, drop or alter messages.

This means that a message may not reach its destination even if the Communication Model states that two nodes can communicate as a matter of principle and the Edge Model lists these two nodes as neighbors. The code below shows the C++ interface for transmission model implementations in Shawn. The send message()-method accepts a MessageInfo data structure containing the message itself, the time of transmission and the position of the sender.
Transmission Model API:

```cpp
struct MessageInfo
{
    Node* src_;  
    Vec src_pos_; 
    double time_; 
    MessageHandle msg_; 
};

void sendMessage(MessageInfo&);
```
5.2.4.2 Sequencer

The sequencer is the control center of the simulation: it prepares the world in which the simulated nodes live, instantiates and parameterizes the implementations of the models as designated by the configuration input and controls the simulation. It consists of Simulation Tasks, the Simulation Controller and the Event Scheduler.

5.2.4.2.1 Simulation Tasks

Simulation Tasks are pieces of code that are invoked from the configuration of the simulation supplied by the user. They are not directly related to the simulated application but they have access to the whole simulation environment and are thus able to perform a wide range of tasks. Example uses are managing simulations, gathering data from individual nodes or running centralized algorithms.

5.2.4.2.2 Simulation Controller

The Simulation Controller acts as the central repository for all available model implementations and runs the simulation by transforming the configuration input into parameterized invocations of Simulation Tasks. In doing so, it mediates between SHAWN's simulation kernel and the user.

5.2.4.2.3 Event Scheduler

SHAWN uses a discrete event scheduler to model time. The Event Scheduler is SHAWN's timekeeping instance. Objects that need the notion of time can register with the Event Scheduler to be notified at an arbitrary point in time. The simulation always skips to the next event time and notifies the registered objects. This process continues until all nodes signal either that they have powered down or until the maximum configured time has elapsed.