# Topology and Time Synchronization algorithms in wireless sensor networks


Víctor López Ferrando

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Memòria del Projecte Final de Carrera

*Enginyeria Informàtica*

Juny 2011

1. Director: Jordi Petit Silvestre

2. President: Maria Josep Blesa Aguilera

3. Vocal: Jaime M. Delgado Merce

Dia de la lectura: 1 de Juliol de 2011

Signatura del tribunal del PFC:

# Abstract

Wireless sensor networks are an active field of research in computer science. The Wisebed project, formed by many European universities, tries to fill the gap between theory and practice, making a platform-independent library of algorithms named Wiselib, and building a testbed in each university accessible through internet. In this project, different topology and time synchronization algorithms are analyzed and implemented into the Wiselib library. These algorithms are later tested through simulation and in the UPC testbed. Theoretical and real life properties of the algorithms are discussed and compared.

---

Les xarxes de sensors inalàmbrics són un camp actiu de recerca en informàtica. El projecte Wisebed, format per diverses universitats europees, intenta omplir el buit entre teoria i pràctica, creant una llibreria d'algorismes anomenada Wiselib i construint xarxes de sensors a cada universitat, accessibles per internet per realitzar proves. En aquest projecte, alguns algorismes de topologia i de sincronització són analitzats i implementats en la llibreria Wiselib. Aquests algorismes es proven mitjançant simulacions i en la xarxa de la UPC. Es discuteixen i comparen propietats teòriques i reals d'aquests algorismes.

A mon pare, ma mare, la meua germana Laura i Anaïs.

# Acknowledgements

I am grateful to Jordi for his support, guidance and confidence from the beggining of the project. I would also like to thank Juan and Antoni for their company and help with the implementation, and also apreciate the help provided by Tobias and Christos with some technical matters.

# Contents

# List of Figures

# List of Tables

# List of listings

# Chapter 1

# Introduction

## 1.1 The project

## 1.2 Wireless sensor networks

Wireless sensor networks (WSN), an active field of research and development, consist of small devices which communicate by radio. It is inherent to this devices to have limited memory and computation resources. They can be applied to solve many different problems and help in many situations, such as industry control, weather monitoring...

In the last years a lot of theoretical WSN research has been done, and many efficient algorithms which take into account the limitations of sensors have appeared. In addition to this, different sensors are available in the market, with different capabilities, operating systems... There is a big gap, however, between the theoretical algorithms (usually tested through simulation), and the real sensors. There is not either any repository of WSN algorithms, or any collaborative platform to develop them.

The Wisebed project (8), started on 2008 and finishing in 2011, aims to develop a free library (Wiselib) (4), which contains many different algorithms implemented in order to work on many platforms. The most relevant algorithms in each category were picked to be added into this library. Moreover, it aims to prepare sensor testbeds on many European universities, to test all these algorithms and collect real world data. This project fits in this framework.

Through this project, many algorithms were studied from their theoretical specification and implemented into the Wiselib, adding other requirements if necessary.

After being implemented, the algorithms are tested through simulation and in real testbeds, such as the one deployed in the Omega building at the Universitat Politècnica de Catalunya (UPC).

## 1.3 Aims of the project

The goal of the project is to implement some Topology and Synchronization algorithms in the Wiselib library and test them through simulation and on real sensors.

Topology algorithms build a subgraph in a network, in order to reduce its complexity, thus reducing energy consumption and interference. The LMST (20) and FLSS (19) were implemented by myself, and they will be described in detail and their implementation discussed in sections 4.2 and 4.3. The KNEIGH (5) and XTC (31) algorithms were implemented by Juan Farré, and I will comment the most important features of each of them on sections 4.4.2 and 4.4.3. It was Juan and I who implemented the first topology algorithms in the Wiselib, decided its more important features and defined their interface. Finally, the CBTC (18) algorithm, implemented by Josep Anguera, is also briefly commented in 4.4.1. In this case I helped with some technical Wiselib implementation matters, and also by implementing some direction interface in shawn simulator (24).

Synchronization algorithms synchronize the sensors' clocks in a network to one or more Base Stations. During the project I implemented and tested the LTS algorithm (29), the TPSN (12) and HRTS (10). Their description and implementation can be found in sections 3.4, 3.5 and 3.6, and the results on the tests in section 3.7.3. For the definition of the interfaces in time synchronization algorithms, including time types, clock... I worked together with Tobias Baumgartner during a coding week on summer 2010.

## 1.4 Wisebed project

My work in this project is developed as part of the Wisebed project (8). The Wisebed project aims to build an infrastructure of interconnected large scale wireless sensor networks for research purposes. The Wisebed project partners work to cover the hardware, software and algorithmic requirements of large scale tests. They are:

- University of Lübeck (Coordinator), Germany

- Freie Universität Berlin, Germany
- Braunschweig Institute of Technology, Germany
- Research Academic Computer Technology Institute, Greece
- Universitat Politècnica de Catalunya, Catalonia
- Universität Bern, Switzerland
- University of Geneva, Switzerland
- Delft University of Technology, Netherlands
- Lancaster University, United Kingdom


**Figure 1.1: Wisebed logo**

The partners in Wisebed have developed or improved applications such as the Tarwis system (13), a web interface to program, configure and manage tests, the WiseML (Wireless Sensor Networks Markup Language), useful to specify test configuration and results, the Wiselib library (4), including dozens of algorithms which run on many platforms, integration with shawn simulator (24)...

Part of the work of UPC in this project has been to develop and integrate many algorithms (topology and time synchronization algorithms) into the Wiselib, and to deploy a testbed formed by iSense sensors and accessible through the Internet to run experiments by any other partner of the project. More details on UPC testbed are given in Section 2.4.

## 1.5 FRONTS project

In parallel to the Wisebed project, there is another European project in the wireless sensor networks field: the FRONTS project (2008-2011). FRONTS is a much more theoretical project, in which studies are made about future network systems emerging in our society. These systems may be wireless, formed by thousands of nodes, changing and self-configuring. The partners in this project are:

- Research Academic Computer Technology Institute (Coordinator), Germany
- Braunschweig University of Technology, Germany
- Universität Paderborn, Germany
- University of Athens, Greece
- Ben-Gurion University of the Negev, Israel
- Università di Roma "La Sapienza", Italy
- Università degli Studi di Salerno, Italy
- Wroclaw University of Technology, Polland
- Universitat Politècnica de Catalunya, Catalonia
- University of Geneva, Switzerland
- University of Lübeck, Germany

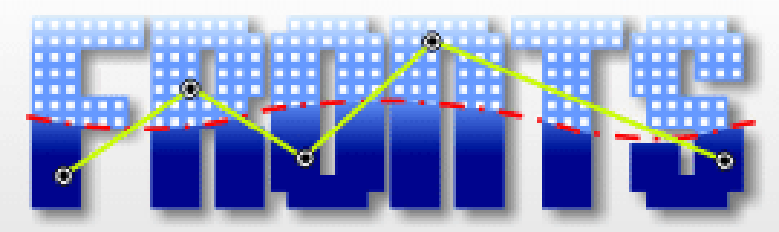


**Figure 1.2: FRONTS logo**

In the last part of the FRONTS project an unifying experiment has been designed, which uses many different modules programmed by each partner. These modules can be grouped in three layers, from bottom to top:

1. **Operating system**: wireless radio, motion of the sensors...
2. **Communication**: nodes are grouped in clusters by clustering algorithms, a highway algorithm builds communication channels between neighbor clusters, and an end-to-end communication algorithm routes messages from any cluster to any other cluster.
3. **Private tracking and data aggregation**: over the communication layer, different applications create safe communication with cryptography, collect data from the whole network, track certain sensors, and plan the motion of mobile sensors in order to improve the tracking.

This experiment was completely integrated into the Wiselib, and as many universities are partners both of Wisebed and FRONTS, there has been a lot of work put in common between both projects.

## 1.6 My work in Wisebed and FRONTS

From June 2009 to June 2011 I have had a grant and worked part-time in the Wisebed project, developing algorithms in the Wiselib. In addition to the pure programming and testing work, I participated in some European meetings, in which general design decisions were made, different parts of the library were put together, and lots of programming was made.

Even though this project explains the work done in the Wisebed project, more specifically in the Wiselib, I also participated in the FRONTS project by attending to some meetings and helping Antoni Segura (colleague from UPC working in FRONTS) with the initial implementations of his algorithm in the Wiselib.

The meetings attended are:

1. WISEBED Technical Meeting (Lübeck, Germany, 21-23/03/2011)
2. FRONTS 3rd Unifying Experiment Workshop (Patras, 24-28/01/2011)
3. FRONTS 2nd Unifying Experiment Workshop (Rome, 23-24/11/2010)
4. PerAda Workshop in Security, Trust and Privacy (Rome, 22-23/11/2010)

5. FRONTS 1st Unifying Exp. Workshop (Braunschweig, 11-15/10/2010)

6. WISEBED Programming Week (Braunschweig, 1-7/08/2010)

7. FRONTS 2nd Winter School (Braunschweig, 12-16/10/2009)

In addition to these meetings, I also attended the Workshop on Software Engineering for Sensor Network Applications (Cape Town, 2-8/05/2010), where I presented Topology control algorithms in WISELIB (2), which describes the development and results of topology algorithms implemented for Wisebed in UPC.

# Chapter 2

# Technology

## 2.1 Wiselib

All the algorithms and programs implemented in this project are part of the Wiselib algorithms library. Wiselib (4, 32) is a generic algorithm library for heterogeneous wireless sensor nodes. This library is under current development within the Wisebed project (8).

Wiselib already covers a large number of algorithmic topics, including routing, clustering, time synchronization, localization, data dissemination, target tracking and topology control. The library is still being enriched by the Wisebed partners. The goal is to develop a library of algorithms for heterogeneous WSNs that is on par with some well-known centralized algorithm libraries existing nowadays (such as LEDA, CGAL or BOOST (6, 7, 14, 16, 27)). This aim has strong requirements:

- Wiselib must run on all sensor nodes by the Wisebed partners and should easily be ported to additional devices.
- Its algorithms should utilize the capabilities of the device they are compiled for.
- Its memory overhead should be as low as possible compared to a native implementation for a specific device.
- Its algorithms should be highly efficient considering the capabilities of the devices.
- Its algorithms implementations should not explicitly deal with platform specific dependencies.

Fig. 2.1 depicts how the Wiselib library is connected with other components of a WSN system. The library can be used by any user application that needs any of the algorithms implemented in it or, alternatively, that needs to implement a new algorithm using the components that the library offers.

The algorithms included in Wiselib itself are organized in topics according to their functionality. In order to abstract the algorithms from the particularities of the physical platform of sensors and the operating system administrating that platform, a set of connectors exists that defines and fixes an interface for interacting with them. A connector is also defined to interact with wireless sensor networks' simulators as, for example, Shawn (24). Those connectors are defined in a way that the same algorithm can be run on a physical platform or on the simulator. Details on how this is achieved are provided in the following.

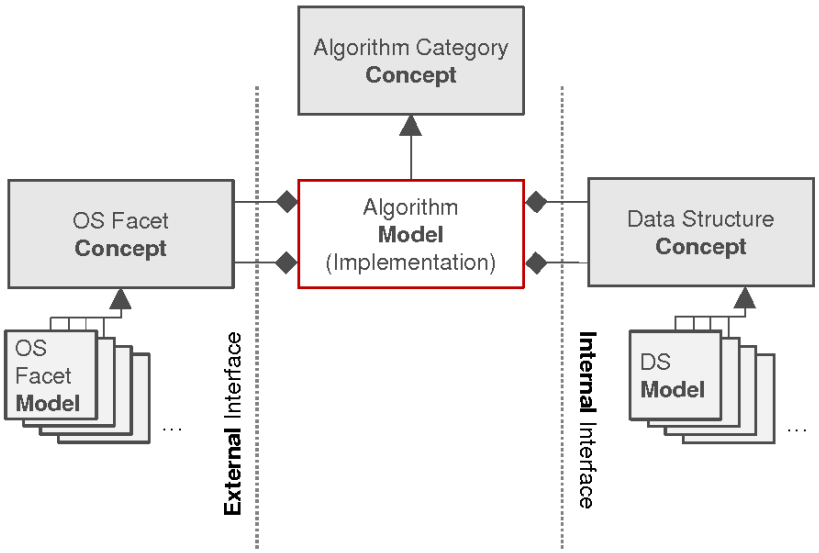



**Figure 2.1: Wiselib architecture** - Wiselib architecture and external interface (32)

### 2.1.1 C++ templates

The programming language chosen to implement and use Wiselib is C++ (1, 28, 30). This decision allows the use of modern programming techniques via object-oriented (OO) design, and provides mostly type-safe development. Moreover, the language offers interesting features such as const-correctness and templates (1, 30). Wiselib mas-

sively uses templates, especially to develop very efficient and flexible applications. The basic functionality of templates is to allow the use of generic code that is resolved by the compiler when specific types are given. Thereby, only the code that is really needed is generated, and methods and parameters as template parameter can be directly accessed. No virtual inheritance is used at all, to avoid the vtables that are necessary to implement virtual function calls. Instead, OO concepts are implemented using template specializations. Using templates and member templates provides the Wiselib with several advantages, such as early binding, inline optimizations, code pruning, extensibility, and a layered structure.

### 2.1.2 Concepts and models

The Wiselib library is not an ordinary object-oriented design with some interfaces and abstract classes. Instead, it uses an efficient generic programming approach that makes extensive use of templates. Thus, an appropriate structure for the description of the several algorithms and components is needed. We heavily employ two generic programming principles (see Fig. 2.2):

*Concepts*. A concept is a detailed description of the requirements for the basic common functionalities of a class of algorithms, that is, an informal prototype for it. For example, later we present the Topology Control concept. Concept do not contain any source code but, instead, are part of the Wiselib documentation. The idea is to provide a complete documentation, in which the concepts and their interrelationships are described, and from which one is able to produce a valid implementation. Interestingly enough, concept inheritance is allowed in Wiselib.

*Models*. A model is a specific implementation of a concept. Any algorithm model implements one or multiple concepts. For example, later we present in detail the concept of a topology control algorithm. With this, one may implement LMST, KNEIGH or any other topology control algorithm: all would be models of the TC concept.

For a more detailed description and examples on the definition of concepts and models, we address to reader to (4).

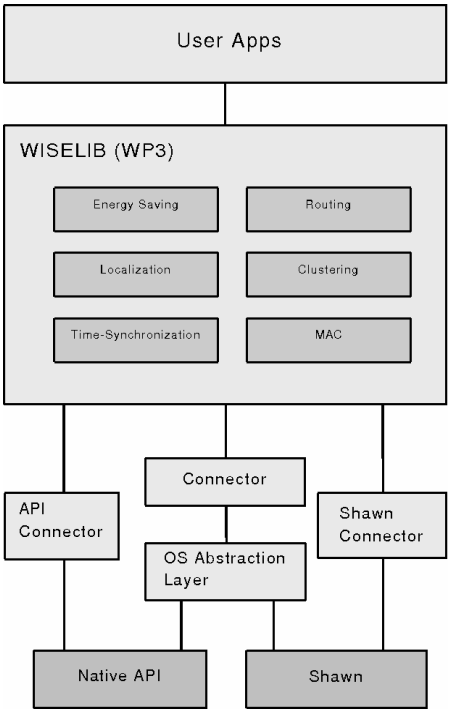



**Figure 2.2: Wiselib external architecture** - Wiselib external architecture (4)

In programming terms, an algorithm model implementing a concept is basically a template expecting various parameters. These parameters can be both Operation System (OS) facets and data structures. OS facets represent the connection to the underlying operating system or firmware (e.g., the sensor radio or the timers), thus being an abstraction layer to the OS, and thus also to the hardware platform. The concepts and models of this abstraction layer form the so-called *external interface* (see Fig. 2.2). OS facets are passed to an algorithm as template arguments and the compiler later resolves such calls to the OS. An OS facet can be implemented by different models, that may have different advantages or purposes. The user can pass any of those models to an algorithm at compile time, and no extra overhead is payed for that. Additionally, the so-named *internal interface* is formed by data structure models and concepts, which make the algorithms independent from specific implementations for their required data structures. As part of this internal interface, Wiselib provides the pMP and the pSTL. The pMP is a C-based implementation of big-number operations. The pSTL is an implementation of parts of the STL that does neither use dynamic memory nor exceptions nor RTTI. The pSTL includes nowadays implementations for the data structures `map`,

`vector` and `list`, and we recently also included `graph` and `priority_queue`. As for the models of the internal interface, the user can pass any of the models for data structures to an algorithm at compile time.

In general, this separation of concepts and their implementation through a template-based approach leads to a highly flexible and powerful design, because the necessities, strength and weaknesses of different hardware platforms or different data structures can easily be utilized just by changing the parameters, or even simply by changing makefile targets.

Thus, an algorithm in Wiselib takes then several template parameters for parameterization. One parameter is the model of the currently used connection component. This way, an algorithm can be compiled for all supported hardware platforms by changing only one parameter. Other parameters may be the kind of container used for storing the neighborhood, or the representation of a general node that is put into the neighborhood container.

Writing extensions for the Wiselib can be interpreted in two different ways. First, a user may want to completely write a new algorithm, which requires to provide a implementation compatible with the corresponding concept. The algorithm can then easily be used with all the other parts of the library. Second, extensibility also covers existing implementations that should be optimized for particular components. E.g., the method to regulate the emission power of a topology control algorithm can be implemented, if a certain node type is used.

The restrictions inherent to wireless sensor networks influenced strongly the design of the Wiselib library and made it a difficult task. As it is known, a WSN environment imposes strong constrains on the resources available for computation, being the most important: (1) the very limited memory and computation power of the tiny microcontrollers where the algorithms will run, (2) the necessity of physical dynamic memory to allow efficient data structures, and (3) the great heterogeneity of the hardware platform. In such a restricted computing environment, efficiency plays an essential role.

The abstraction of the OS and hardware platform provided by the external interface is used by Wiselib to provide single implementations of its algorithms that can be run on heterogeneous test beds. In its current version, Wiselib supports nine different sensor types, namely, iSense nodes, SCW MSB, SCW ESB, Timote Sky, MicaZ,

| Hardware | Firmware/OS | CPU | Language | Dynamic Memory | ROM Size | RAM Size | Bits |
|---|---|---|---|---|---|---|---|
| iSense | iSense-FW | Jennic | C++ | Physical | 128kB | 92kB | 32 |
| SCW MSB | SCW-FW | MSP430 | C | None | 48kB | 10kB | 16 |
| SCW ESB | SCW-FW | MSP430 | C | None | 60kB | 2kB | 16 |
| Tmote Sky | Contiki | MSP430 | C | Physical | 48kB | 10kB | 16 |
| MicaZ | Contiki | ATMega128L | C | Physical | 128kB | 4kB | 8 |
| TNOde | TinyOS | ATMega128L | nesC | Physical | 128kB | 4kB | 8 |
| iMote2 | TinyOS | Intel XScale | nesC | Physical | 32MB | 32MB | 32 |
| GumStix | Emb. Linux | Intel XScale | C | Virtual | 16MB | 64MB | 32 |
| Desktop PC | Shawn | various | C++ | Virtual | unlimited | unlimited | 32/64 |
| Desktop PC | TOSSIM | (ATMega128L) | nesC | (Physical) | unlimited | unlimited | (8) |

**Table 2.1:** Wiselib target platforms (4)

TNOde, iMote2, GumStix, and desktop PCs. Each of these sensor nodes have a different micro-controller type, its own operating system, its more prominent programming language, its kind of dynamic memory, its amount of ROM and RAM, and its bit width (see Table 2.1). Observe that two simulator platforms are also considered (Shawn and TOSSIM).

## 2.2 Shawn

Shawn (24) is Wireless Sensor Networks simulator which will be extensively used in this project. The simulator is of great utility in the development of the algorithms, because it offers a fast way of testing algorithms' correctness. This simulator will also be used to perform large scale experiments (with hundreds or thousands of sensors), for which no real scenario is available.

The Shawn simulator is implemented in C++ and is extremely fast when running the experiments. It offers many configuration options when running the tests, the most commonly used are:

- Number of sensors
- Size of the scenario
- Range of the radio of the sensors
- Probability of package loss

The experiments take place in an scenarios whose shape can be chosen: rectangular, circular... The sensors can be located at random if desired, but can also be loaded from an XML file. This second possibility is specially helpful when comparing the results of different algorithms.

The simulator prints through the standard output the debug messages of sensors, and we will use this output to draw the results of topology control algorithms. Using a visualizer written in Python by Antoni Segura we will obtain nice-looking images of the resulting topology after running the algorithms.

## 2.3 iSense

The simulator is useful, but our main goal is to run algorithms in real sensors, and the ones used will be iSense. iSense sensors are commercialized by the company Colasenses (9), and their main characteristics are listed in Table 2.1.

**Figure 2.3: iSense core module** - contains the processing unit, radio, timer and clock

iSense sensors consist of a core module (see Fig. 2.3), to which many other modules can be attached. These other modules (see Fig. 2.4) are such as the gateway module (USB connection to PC), used for flashing and debug logging, battery and solar power system, GPS module, and many other measurement modules.

These sensors are programmed in C++ and compiled with ba-elf-c++ compiler for the JENNIC processor. It has a closed source operating system, which will provide all the necessary calls for sending and receiving messages, checking and changing time, setting transmission power, determining location and time through the GPS module...

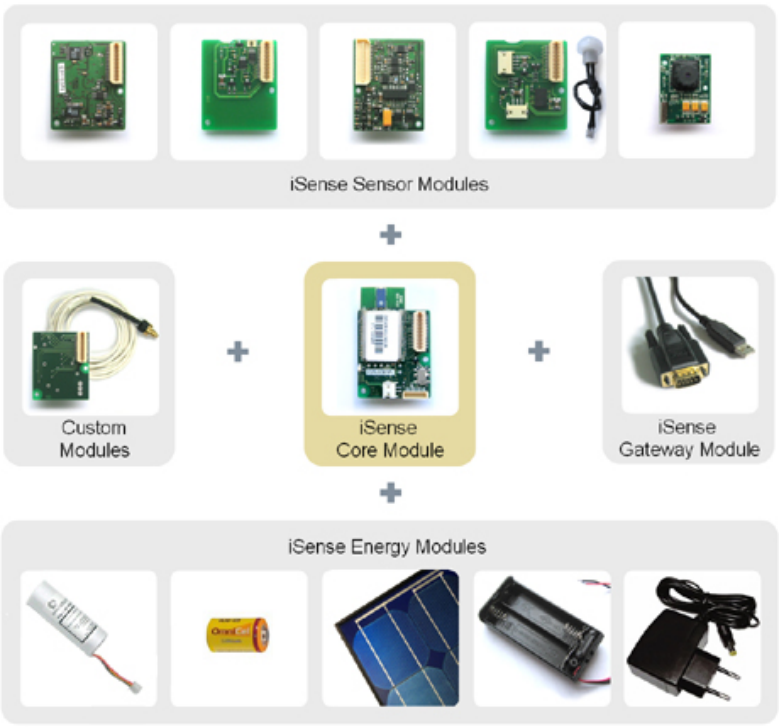



**Figure 2.4: iSense extension modules** - includes measurement, communication and energy modules

But we will not implement iSense applications. Instead, our applications are integrated in the Wiselib, which provides common interfaces for using all the previous functionalities for all platforms. We will have to integrate any functionality required into the Wiselib, instead of using the iSense calls directly, thus maintaining the platform independence of the algorithms and applications.

Our Wiselib applications, using Wiselib algorithms can be compiled for iSense and loaded to the sensors using iShell (see Fig. 2.5), a Java application used to flash sensors, run the programs and collect the debug messages. This same application is used to monitor the execution of the programs.

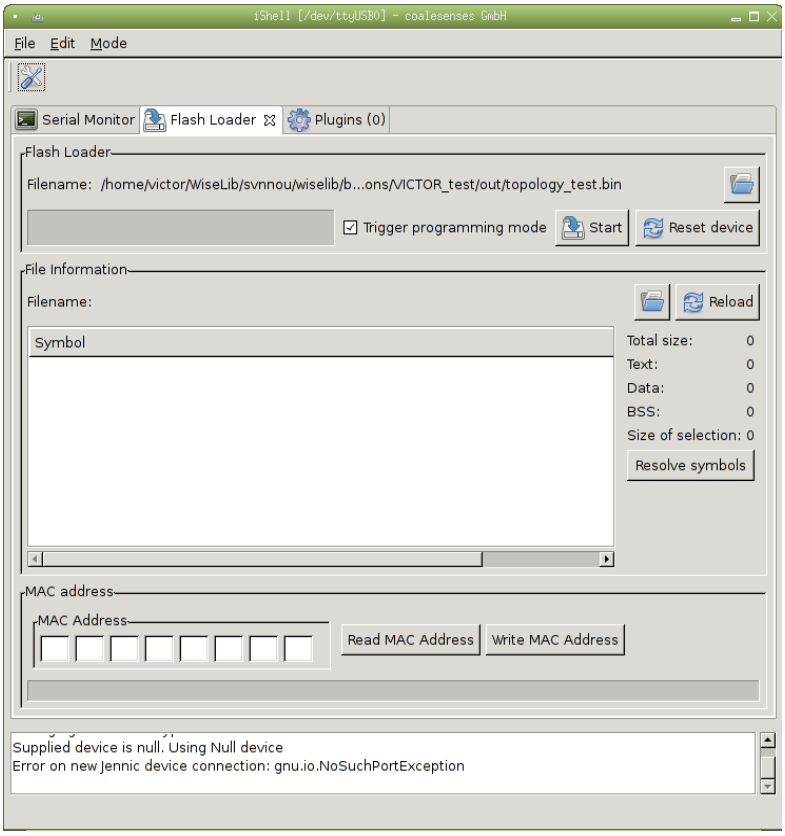



**Figure 2.5: iShell** - Java application for flashing sensors and collecting debug messages

## 2.4 Testbed

The tests on real nodes take place in testbeds, which are sets of nodes deployed in some building, and to which access is provided via a web interface. These testbeds also include software to collect all debug messages sent, giving as a result a trace of all logged messages sent during the experiment.

The Wisebed infrastructure consists of several federated test beds of various sizes (dozens to hundreds of nodes), currently including nine test beds that amount to 600 nodes. This infrastructure is inherently heterogeneous, using hardware with different computational resources and sensing capacities (see Table 2.1). The Wisebed federated test beds are interconnected together using the TARWIS system (13), which provides a web-based GUI for test bed management, experiment configuration, and experiment monitoring.

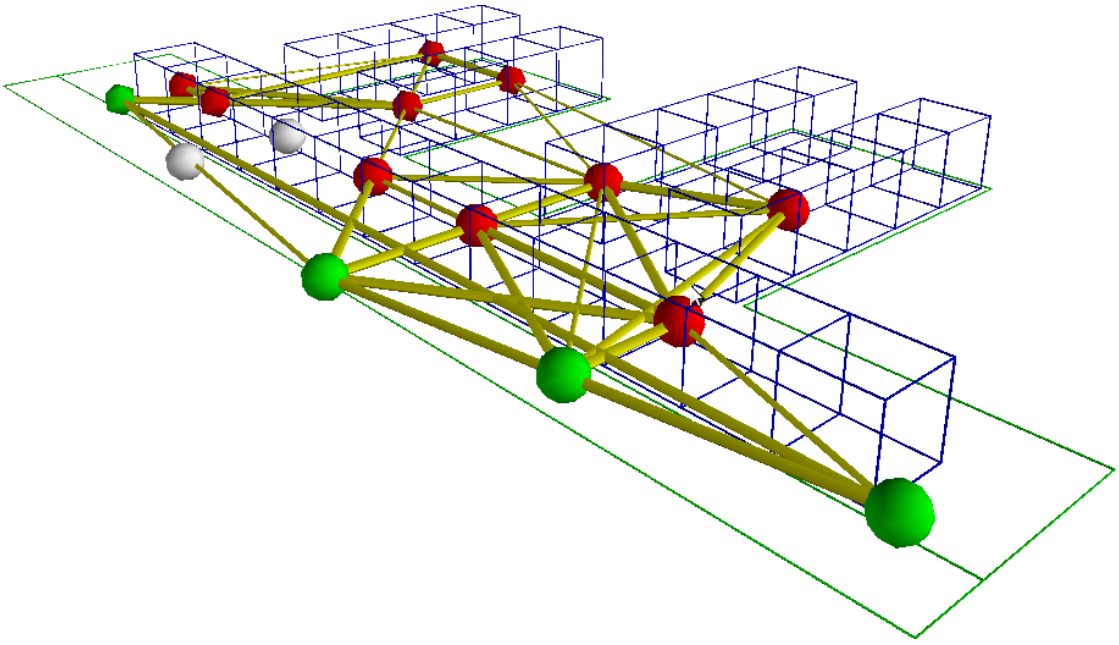


**Figure 2.6: UPC Testbed** - Representation of the UPC test bed. The width of the links is proportional to their measured capacity at full power. Failing links are not shown.

The UPC testbed consists of 17 iSense nodes from Coalesenses GmbH. The iSense nodes use a Jennic JN5139, a solution that combines the controller and the wireless communication transceiver in one chip. The controller has a 32-bit RISC architecture

and runs at 16MHz. It offers 96kB of RAM and 128kB of Serial Flash. It has a transmit power from -60dBm to +3dBm, which reaches ranges of up to 500m in the free.

The 17 nodes are located in our departmental building. In this test bed, sensor nodes in the entire network are not reachable by each other. Because of the contrived architecture of our building, their range is strongly shortened and some sensors must communicate with others using multiple hops. Fig. 2.6 depicts the structure of our building, the location of the nodes and their topology at full power. This representation is not the current disposition of the sensors, but the one they had when the tests were run.

Topology algorithms were tested on the UPC testbed at the beggining of 2010, at which time the testbed had the form in Figure 2.6. The results, obtained through debug messages sent to each sensors' attached computer will be used to draw in the 3D visualizer seen in Fig. 2.6, written in Python and developed by Jordi Petit.

# Chapter 3

# Time synchronization

## 3.1 Time synchronization algorithms

Time synchronization algorithms (25, 34), aim to synchronize the clocks of the sensors in a network via radio communication. Sensors' clocks start when they are turned on, and are not very accurate. These algorithms will synchronize the clocks to one or more reference nodes. These nodes might be equipped with a GPS module, which provides GMT time, if real time is needed.

Time synchronization is important if any application needs the sensors to execute a certain task at some time, or if some external events need to be logged together with the time when they happened. If synchronization is accurate enough, some applications might even use the timestamp of some events such as a noise, to triangulate the position of the source of the noise.

Time synchronization algorithms can be classified according to different properties, such as:

- **Global versus local algorithms**: a global algorithm tries to keep all sensors in the network synchronized, while a local algorithm only synchronizes sensors locally (some logical part of the network, a broadcast domain...)
- **Hardware- versus software-based algorithms**: some algorithms require specific hardware such as a GPS module, a dedicated radio channel, or high precision clocks in order to perform as expected, while software-based algorithms only use the standard radio and message exchange.

- **A priori versus a posteriori synchronization**: in a priori algorithms, the network is synchronized all the time. On the other hand, a posteriori synchronization runs after some external event has taken place.

In order to compare different synchronization algorithms, it is important to know what performance metrics have special relevance. These are the most important ones:

- **Precision**: the error of the synchronization, between the reference node and the rest of the network.
- **Energy costs**: which can be stated by the number of messages sent by the algorithm.
- **Memory requirements**: if a history of previous synchronizations is kept, future synchronizations may be more precise, but more memory will be spent.
- **Fault tolerance**: how does the algorithm perform in cases such as node failure, package loss, etc.

## 3.2 Sender-receiver and receiver-receiver algorithms

Time synchronization algorithms can be divided in two main types: sender-receiver and receiver-receiver algorithms. Starting from a reference node, to which we want to synchronize the rest of the network, this can be achieved by:

- **Sender-receiver**: the reference node (sender) communicates to each other sensor (receiver), and after some message exchange the receiver is synchronized.
- **Receiver-receiver**: the reference node (sender) sends a message, and the synchronization takes place between two receivers, who compare their timestamps of the same message.

### 3.2.1 Sender-receiver schema

In the synchronization of sensors $i$ and $j$ using a sender-receiver schema, three messages are exchanged, two of them contain timestamps, and the third one notifies the receiver its offset:

1. The sender (reference node) timestamps a message (adds the current time to a message), which will be $t_0$.

2. The message is sent, and received by the receiver. The receiver stores a timestamp $t_1$ at which it received the message. Note that theoretically $t_1 = t_0 + \tau_1 + \delta_1 + \eta_1 +$ offset, where $\tau_1$ is the message processing before it is sent, $\delta$ is the propagation time, and $\eta$ is the message processing before the interruption is executed.

3. The receiver does whatever is required by the algorithm, then timestamps the message again with $t_2$, and sends it back to the sender.

4. The sender timestamps the message when received, namely $t_3$, and then calculates the offset between sender and receiver's clock. Using the same variables as before, the equation is: $t_3 = t_2 + \tau_2 + \delta_2 + \eta_2 -$ offset. The assumption we will do, and the source of error in this method, is that $\tau = \tau_1 = \tau_2$, $\eta = \eta_1 = \eta_2$ and $\delta_1 = \delta_2 = 0$. So we have that:

$$\tau + \eta = t_1 - t_0 - \text{offset} = t_3 - t_2 + \text{offset}$$

$$\text{offset} = \frac{t_0 + t_3 - t_1 - t_2}{2}$$

5. After the offset is calculated by the sender, a message is sent to the receiver, to inform it of its offset, and so it adjusts its clock.

The error in this method comes from $\tau + \eta + \delta$, which is the time it takes the operating system to process a message, send it

### 3.2.2 Receiver-receiver schema

In this receiver-receiver algorithms, sensors don't synchronize with the sender of a timestamp message, instead they synchronize with other receivers of the same message.

Let's imagine an scenario with three sensors: $n_1$, $n_2$ and $n_3$, and we want to synchronize $n_1$ and $n_2$:

1. $n_3$ sends a message, which is timestamped by $n_1$ ($t_1$) and $n_2$($t_2$).

2. $n_1$ sends $n_2$ its timestamp $t_1$.

3. $n_2$ calculates its new time as: $T = t - t_2 + t_1$.

In this case, the assumption made is that $n_1$ and $n_2$ timestamped the message at the same time, and the uncertainty comes from $\eta$, the time between the message is received, and the interruption is executed. The receiver-receiver method is thus more accurate than the sender-receiver.

We will see an example of a receiver-receiver algorithm in section 3.6, in which the HRTS algorithm is described.

## 3.3 Graph algorithms

### 3.3.1 Introduction

Our first synchronization algorithm is LTS (29), which depends on a graph algorithm to spread the synchronization to the whole network. In this section we will introduce two graph algorithms which will later be used by LTS.

Graph algorithms, when run on a Wireless Sensor Network, build a graph, taking sensors as nodes. These algorithms are required by LTS algorithm to expand the synchronization to all sensors in the network. In (29) the DDFS (3) is proposed to build a tree starting from a reference node. We also chose to implement DBFS (35), which minimizes the depth of the tree, meaning less synchronizations will take place between the reference node and the leaves of the tree, thus reducing the error.

In this section we will see how the DDFS and DBFS algorithms work, and how they were implemented into the Wiselib.

### 3.3.2 DDFS: Distributed Depth First Search

The Distributed Depth First Search algorithm (3), builds a tree, having as root one node, and doing the search in a depth-first fashion. It builds a generic DFS tree, but it works in a totally distributed way, meaning every sensors only needs information about its broadcast domain for building the tree correctly.

#### 3.3.2.1 Description of the algorithm

In DDFS, as in DFS, the search is centered in one node at each moment in time. When one node is the center, all the neighbors which have not yet been visited, are visited, and

when none is left, the search is returned to the parent. In DDFS, some data structures will also be used by each node in order to know which nodes have already been visited.

Four different messages will be sent:

- `Discover`: messages arriving to a node when it is visited for the first time.
- `Return`: message returning the center to a node.
- `Visited`: sent by a node when it is visited for the first time.
- `Ack`: acknowledgement, sent as response to a visited message.

Variables kept at node i:

- `Neighbors(i)`: set of neighbors of each node.
- `Father(i)`: the father of node *i* in the DFS tree. Output of the algorithm.
- `Unvisited(i)`: the subset of `Neighbors(i)` , which contains the neighbors from which the `Visited` message has not been received.
- `Flag(i, j)`: a binary flag, initially set to 0, which equals 1 in the interval after `Visited` was sent and before `Ack` is received.

The algorithm starts the search by the root sending to itself a `Return` message. We can describe the rest of the algorithm by analyzing how it works when receiving each kind of message:

- `Discover`: send a `Visited` message to all the neighbors except to the father. Set the `Flag` to 1 and send a `Return` to `father` if it has no neighbors.
- `Return`: if there are unvisited nodes, send a `Discover` to the first one and remove it. If all the neighbors are visited, return the search to your father. In case you are your father (root), stop the algorithm.
- `Visited`: send an `Ack` to the node which has been visited.
- `Ack`: set the `Flag` to 0. If all the `Ack`s have been received, deliver a `Return` to yourself and resume the search.

In figure 3.1 we can see an example of how the DDFS algorithm behaves having three nodes. This figure shows the messages exchanged and the resulting graph.

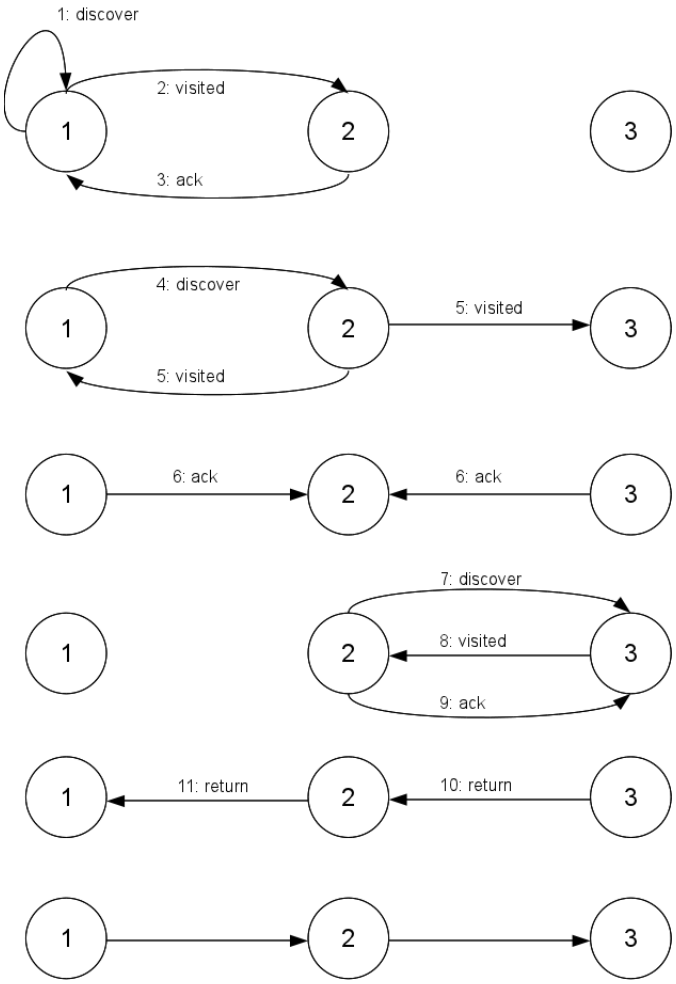



**Figure 3.1: DDFS example** - Example execution with three nodes. The last graph is the result of the algorithm

#### 3.3.2.2 Implementation

In this section we will take a look at the most important aspects in the implementation of DDFS in the Wiselib. In its implementation, a preliminary phase needed to be added: the neighborhood discovery. DDFS takes as input the neighbors of each node, but this information is not known when the algorithm starts. We added then a first phase during which broadcast messages are sent, and the neighbors are stored. Once the neighborhood is known, DDFS can start. Let's now see how it is implemented.

As in all algorithms in the Wiselib, the class will have template parameters:

```
template<typename OsModel_P,
         typename Radio_P = typename OsModel_P::Radio,
         typename Debug_P = typename OsModel_P::Debug,
         uint16_t MAX_NODES = 32>
class DdfsGraph
{...
```

**Listing 1: DDFS template parameters** - Operating system, radio, debug channel and maximum number of nodes.

First we have the operating system model, which will vary if compiled for different platforms. The next two parameters are the radio and the debug channel, which by default are derived from the OsModel. Finally there is an integer with the maximum number of nodes in the network, used for allocating the memory of the data structures statically, in case the target platform doesn't support dynamic memory allocation.

We also define the message ids, note that we have one for each of the messages listed before:

```
enum DdfsGraphMsgIds {
   DdfsMsgIdDiscover = 130, // DISCOVER messages
   DdfsMsgIdReturn = 131, // RETURN messages
   DdfsMsgIdVisited = 132, // VISITED messages
   DdfsMsgIdAck = 133, // ACK messages
   DdfsMsgIdNeighbourhood = 134, // NEIGHBORHOOD messages
};
```

**Listing 2: DDFS message ids** - `Discover`, `Return`, `Visited`, `Ack` and `Neighborhood`.

And these are the private data structures needed by the algorithm:

```
bool set_ddfs_delegate_;
ddfs_delegate_t ddfs_delegate_;

bool root_;
millis_t startup_time_, neighbourhood_construction_time_;

vector_static<OsModel, node_id_t, MAX_NODES> neighbours_;
vector_static<OsModel, node_id_t, MAX_NODES> unvisited_;
vector_static<OsModel, bool, MAX_NODES> flag_;
```

**Listing 3: DDFS data structures** - a delegate (see A.1.2), `root`, which indicates if the sensor is root of the tree, two timers to delay the start and limit the neighborhood discovery phase, and three vectors which contain the variables needed, as explained in the description.

It is very important to note the variable `ddfs_delegate_` (the use of delegates is further explained in A.1.2). This variable is used to store the pointer to a function which will be called when the algorithm stops. We are planning on using this algorithm to propagate through the network the synchronization of sensors, so it is very important to have the capability of getting a notification when the algorithm has finished running.

This is the definition of `ddfs_delegate_t` and the function to register the callback:

```
typedef delegate0<void> ddfs_delegate_t;

template<class T, void (T::*TMethod)()>
inline void reg_finish_callback( T *obj_pnt )
{
   ddfs_delegate_ = ddfs_delegate_t::from_method<T, TMethod>( obj_pnt );
   set_ddfs_delegate_ = true;
};
```

**Listing 4: DDFS register callback** - registers callback functions, which are called when the algorithm finishes.

The most important function in this algorithm, which controls all the behaviour of the algorithm is the `receive()` function. As we could see in the description of the

algorithm, the running of DDFS can be detailed by how it responds to each message. The `receive()` function receives calls from the radio, modifies the needed variables, and sends the corresponding messages. The implementation of `receive()` cannot be found in the Program code 38 in page 102.

#### 3.3.2.3 Simulations

The easiest way of checking the correctness of this algorithm is by simulating an application which uses it, and drawing the tree built by the algorithm.

We can see the results in the simulation in Figures 3.2 and 3.3, generated by giving the visualizer written by Antoni the output of a Shawn simulation. In these images we see 30 and 100 nodes respectively (the root is bigger), purple lines representing possible communication (nodes can hear each other), and in yellow the resulting DDFS graph. It is clear that the yellow edges build a tree which has a DFS-like shape (very deep).

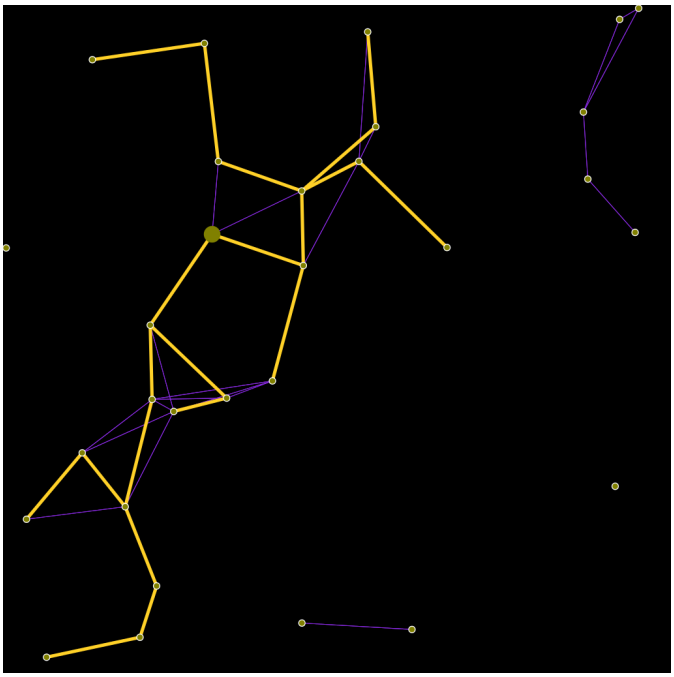


**Figure 3.2: DDFS 30 nodes** - DDFS simulation results with 30 nodes

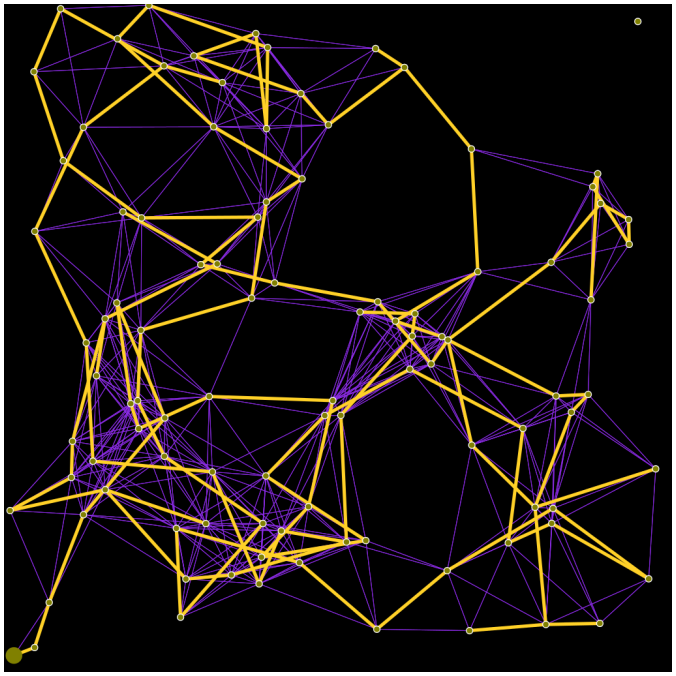


**Figure 3.3: DDFS 100 nodes** - DDFS simulation results with 100 nodes

### 3.3.3 DBFS: Distributed Breadth First Search

When seeking a way of propagating the synchronization in the network through a tree with the reference node in the root, it is obvious that it is desirable to reduce the tree's height. At every level of the tree the error gets bigger, and so it is preferred to do as less synchronizations as possible from the root to the leaves.

For this reason we will implement the Distributed Breadth First Search algorithm (35), in addition to DDFS. Note that this algorithm doesn't rely on any distance estimation, but reduces the number of hops from the root to the leaves. Just as DDFS, it works on a totally distributed manner, which we will detail in the next sections.

#### 3.3.3.1 Description of the algorithm

This algorithm builds a BFS tree by exploring the network, starting from the root, one level at a time. Each iteration starts with a labelling phase, in which `Label` messages are sent to the BFS tree children. When the labelling phase reaches the leaves, it still broadcasts one more message. The neighbors who listen to these messages, are the new

leaves, and notify the root with the Echo phase, in which `Echo` messages are sent to the root. The root decides when the search ends.

The two kinds of messages are:

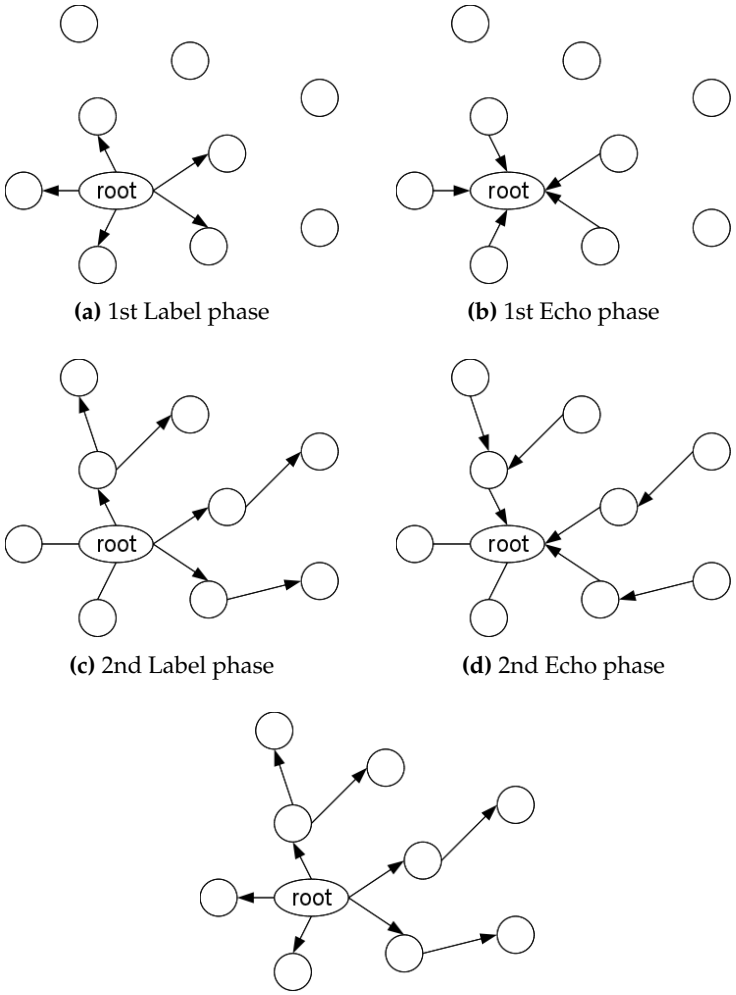
- `Label(lev)`: sent to notify the neighbors to label themselves with level `lev+1`.
- `Echo(status)`: a response message sent up to the tree. The status parameter can take the values:
  - `keepon`: the sender still has neighbors which may not have labels.
  - `stop`: sender doesn't want to receive any more `Label` messages, because it cannot become child.
  - `end`: sender doesn't have unexplored neighbors, so the search is ended in its branch.

And the local variables for node $i$ are:

- `Neighbor`: set of neighbor nodes.
- `Labeled`: boolean value, true if and only if node $i$ is labeled.
- `Parent`: the parent of $i$ in the BFS tree. Root is its own parent.
- `Level`: if `Labeled`, the level in the tree.
- `SendTo`: subset of `Neighbor` to which node $i$ sends `Label` messages.
- `Child`: children of $i$ in the BFS tree.
- `Echoed(j)`: variable with value false if a `Label` message was sent to node $j$, but no `Echo` message was received yet.

In Figure 3.4 we can find an example of how the algorithm works. The DBFS algorithms runs steps, and at each step, one level of the BFS tree is added. Each step can be divided in two phases: Label and Echo. In the Label phase, messages are sent to the leaves until one more level is discovered.

In the Echo phase, the results are notified upwards. Echo messages specify if the search is finished or can continue, and the root will decide to stop the algorithm once it knows all the nodes were labeled. Note from the third subfigure, that the search is not continued through the branches that have been fully labeled.




**(a)** 1st Label phase

**(b)** 1st Echo phase

**(c)** 2nd Label phase

**(d)** 2nd Echo phase

**(e)** DBFS graph

**Figure 3.4:** Execution in parallel of DBFS. After a Label and an Echo phase, one level of the tree is built.

#### 3.3.3.2 Implementation

The DBFS implementation is very similar to the one of DDFS. DBFS also starts with a neighborhood discovery phase, both have the same template parameters, and the same callback functionality.

The message ids used in this algorithms are:

```
enum DbfsGraphMsgIds {
   DbfsMsgIdLabel = 130, // LABEL messages
   DbfsMsgIdEcho = 131, // ECHO  messages
   DbfsMsgIdNeighborhood = 132, // MNEIGHBORHOOD messages
   EchoKeepon = 0, // ECHO KEEPON messages
   EchoStop = 1, // ECHO STOP messages
   EchoEnd = 2, // ECHO END messages
};
```

**Listing 5: DBFS message ids** - `Label`, `Echo`, `Neighborhood`, `Keepon`, `Stop` and `End` messages.

As in DDFS, the most important function is `receive()`, which handles all the different states of the algorithm. Its implementation can be found in Program code 39 on page 105.

#### 3.3.3.3 Simulations

Such as in DDFS, the most effective way of testing this algorithm is by simulating it and drawing the resulting BFS tree. A BFS tree is very easy to recognize, because it will be clear which node is the root and the levels will grow in number of nodes very rapidly. We can see the results of some simulations in Figures 3.5, 3.6,3.7 and 3.8.

## 3.4 LTS: Lightweight Time Synchronization

The Lightweight Time Synchronization (29) is a simple algorithm, based in the Sender-receiver schema. The algorithm has two parts:

1. A tree is built, having as root the reference node.

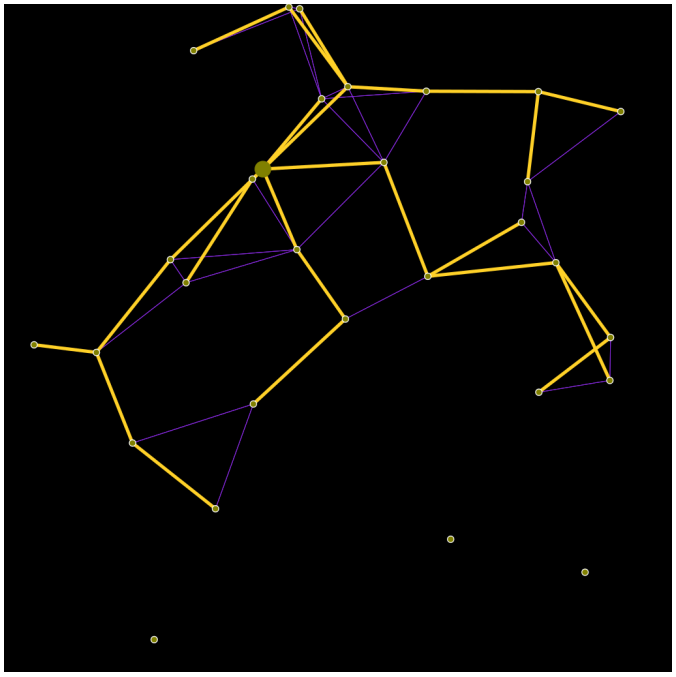


**Figure 3.5: DBFS 30 nodes** - DBFS results with 30 nodes

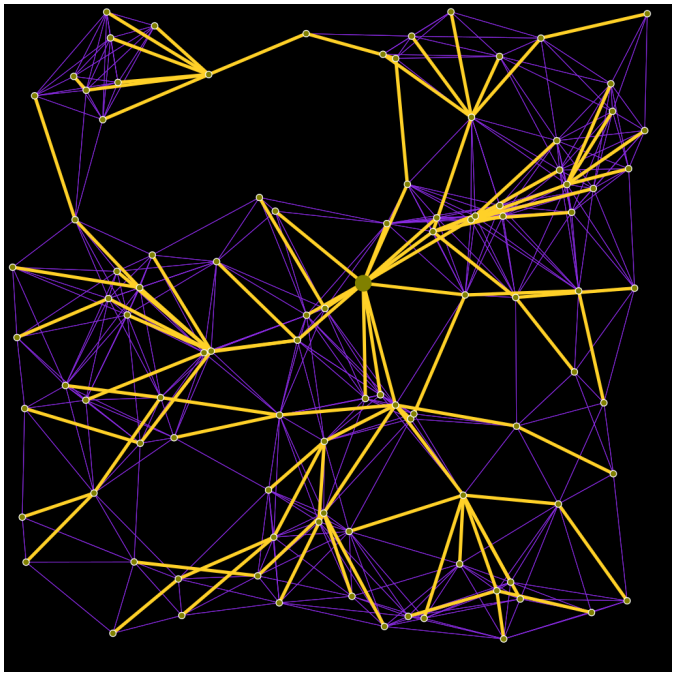


**Figure 3.6: DBFS 100 nodes** - DBFS results with 100 nodes

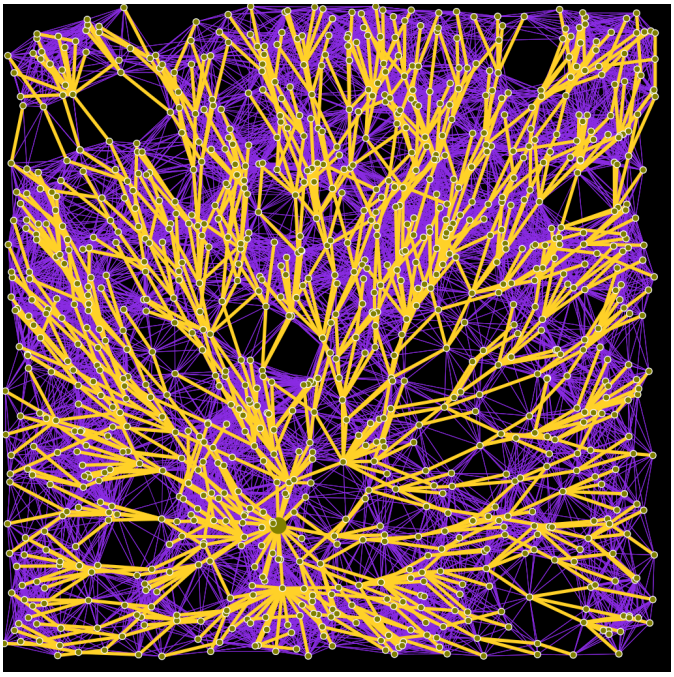


**Figure 3.7: DBFS 1000 nodes** - DBFS results with 1000 nodes

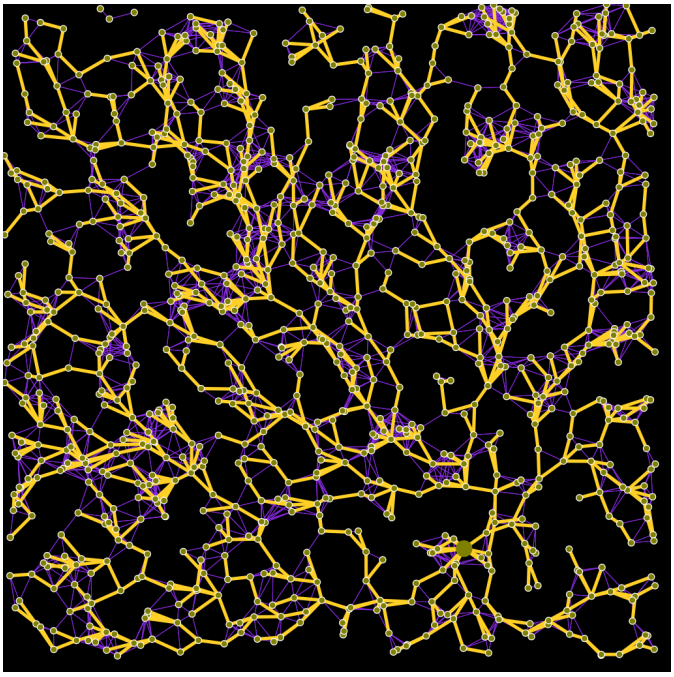


**Figure 3.8: DBFS 1000 nodes** - DBFS results with 1000 nodes and less radio range

2. Synchronization takes place from the root to the rest of nodes through the edges of the tree.

### 3.4.1 Description of the algorithm

Having the algorithm running on every sensor, the first step is to start the tree construction from the reference node. In (29) the DDFS algorithm is used, but we chose to use also the DBFS, as it reduces the tree depth and so decreases the error.

Once the tree is built, the synchronization starts in a sender-receiver fashion (see section 3.2.1). In every edge of the tree, three messages are exchanged:

1. **Reference** $\rightarrow$ **Node**: first sender-receiver message, with one timestamp.
2. **Reference** $\leftarrow$ **Node**: second sender-receiver message, with three timestamps.
3. **Reference** $\rightarrow$ **Node**: offset notification to node.

We can also visualize the pair-wise synchronization in Figure 3.9.

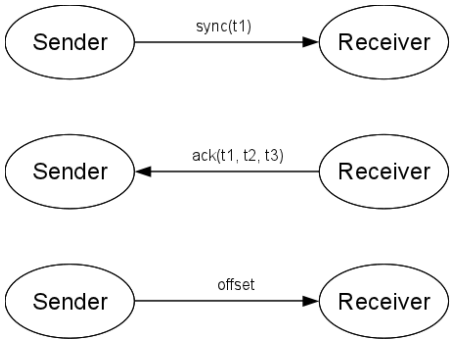



**Figure 3.9: LTS pair-wise synchronization** - synchronization of a pair of nodes using the sender-receiver schema

When the node receives its offset from the reference node, it synchronizes its clock. Having its clock synchronized, it will proceed to synchronize its children in the tree, unless it is a leaf node, in which case its work is finished.

### 3.4.2 Implementation

We will detail in this section the implementation of the LTS algorithm, looking into how the tree is built by an algorithm passed as a template parameter, the definition as

a subclass of `SynchronizationBase` class, the definition of a message class, and how the clock is handled. The explanation of the implementation of other synchronization algorithms will overlook this aspects, as work in the same way.

#### 3.4.2.1 The SynchronizationBase class

The `SynchronizationBase` class, parent of all the synchronization algorithm classes, is a class which implements the callback handling (further explanation in A.1.2). Many algorithms and applications in the Wiselib depend on other algorithms: we can imagine an application that collects data from them weather, which will start working once the nodes are synchronized, not before. This can be managed by callbacks: the application registers a function which will be called once the sensor is synchronized, and will start the data collection.

This class implement the next functions, used by other algorithms or applications:

```
template<class T, void (T::*TMethod)()>
void reg_listener_callback( T *obj_pnt )
{
   callback_ =
         synchronization_delegate_t::from_method<T, TMethod>( obj_pnt );
}
```

**Listing 6: SynchronizationBase register callback** - define the callback to be called when the synchronization has taken place.

```
void unreg_listener_callback( void )
{
   callback_ = synchronization_delegate_t();
}
```

**Listing 7: SynchronizationBase unregister callback** - unregisters an already registered callback.

```
void notify_listeners()
{
   if (callback_)
      callback_();
}
```

**Listing 8: SynchronizationBase notify listeners** - calls the callback if registered.

Using these three functions, a very important functionality is added to synchronization algorithms: they can notify other algorithms and applications when the synchronization has taken place.

#### 3.4.2.2 LTS message class

A class is defined to handle the synchronization messages used to send timestamps. It's header is:

```
template<typename OsModel_P,
         typename Radio_P,
         typename time_t_P>
class LtsSynchronizationMessage
```

**Listing 9: LTS message class header** - it uses the Radio to get the block data type, and the time type to reserve the memory for the buffer.

Its template parameters are the operating system model, the radio and the time type. It uses the block data type (found in the radio) and the time type to declare a buffer used for messages containing timestamps:

```
block_data_t buffer[1 + TIME_SIZE*3];
```

**Listing 10: LTS message buffer** - The bigger messages contain an id and three timestamps.

This buffer can be filled with some auxiliary get and set functions as:

```
inline void set_t1( time_t t1 )
{ memcpy( buffer + TIME_POS, &t1, TIME_SIZE ); }

inline time_t t1()
{
   time_t t1;
   memcpy( &t1, buffer + TIME_POS, TIME_SIZE );
   return t1;
}
```

**Listing 11: LTS message get and set functions** - functions to set and get the first timestamp.

The `LtsSynchronizationMessage` class will help `LtsSynchronization` algorithm with message handling, and is useful to ease the code in some of the algorithms' functions.

#### 3.4.2.3 Graph algorithm as a template parameter

An important decision on the implementation of LTS was to have a graph algorithm as a template parameter:

```
template<... ,
        typename NeighborhoodDiscovery_P =
                typename wiselib::DdfsGraph<OsModel_P,
                                            Radio_P,
                                            Debug_P,
                                            MAX_NODES>,
        ...>
class LtsSynchronization ...
```

**Listing 12: LTS graph algorithm template parameter** - the graph algorithm used in LTS is provided as a template parameter.

(29) stated the DDFS to be used, but also the DBFS could be used, improving the performance. This leaded to the decision of creating a graph algorithm concept, and requiring it as a template parameter for this algorithm. This template parameter, which has the DDFS algorithm as a default, will be used for the tree construction, as we can see:

```
neighborhood_discovery_.set_root( root );
neighborhood_discovery().template
   reg_finish_callback<self_type,
                       &self_type::start_synchronization>( this );
neighborhood_discovery_.reinit();
```

**Listing 13: LTS call to the graph algorithm** - initialization of the graph algorithm: tells the graph algorithm if it is the root of the tree, registers a callback function to be called when the tree is built and restarts the graph construction.

These previous lines of code do the following:

1. Set the node to be or not the root of the tree.
2. Register as a callback the function `start_synchronization()`, which will be called once the tree is built.
3. Restart the tree construction.

#### 3.4.2.4 Clock handling in the WISELIB

Some algorithms only need the time to timestamp events, but because synchronization algorithms need to operate with time: add, divide... an extended time concept, which implements all these operations was created. A piece of the extended time implementation for iSense is as follows:

```
iSenseExtendedTime operator+(const iSenseExtendedTime& exttime)
{
   return isense::Time(*this) + isense::Time(exttime);
}
iSenseExtendedTime operator+=(const iSenseExtendedTime& exttime)
{
   *this = *this + exttime;
   return *this;
}
```

**Listing 14: ExtendedTime operators** - some of the operators in `ExtendedTime` class for iSense.

It is clear that the type which we will use to handle the time will be this Extended-Time.

The clock we need isn't either a normal clock. In synchronization algorithms we don't only need to know the time, but also be able to modify it's time. Thus our algorithm needs a clock which implements the SettableClock concept.

This concept will provide us the necessary functions: `time()`, `seconds()`, `milliseconds()`, `microseconds()`, `state()` and `set_time(time_t time)`. It is important to note that this clock can use different time definitions (could be a simple one, or the one that implements arithmetic operations), so it has the time as a parameter. For example in iSense:

```
template<typename OsModel_P,
         typename Time_P = isense::Time>
class iSenseClockModel
{
```

**Listing 15: iSense clock header** - template parameters of the iSense clock interface, has the time type in which the queries will be returned.

#### 3.4.2.5 Implementation of LTS

In our implementation of LTS, lets first have a look at the template parameters:

```
template<typename OsModel_P,
         typename Radio_P = typename OsModel_P::Radio,
         typename Clock_P = typename OsModel_P::Clock,
         typename Debug_P = typename OsModel_P::Debug,
         typename NeighborhoodDiscovery_P =
            typename wiselib::DdfsGraph<OsModel_P,
                                        Radio_P,
                                        Debug_P,
                                        MAX_NODES>,
         uint16_t MAX_NODES = 32>
class LtsSynchronization : public synchronizationBase<OsModel_P>
{
```

**Listing 16: LTS template parameters** - operating system, radio, clock, debug channel, graph algorithm and maximum number of nodes are provided through template parameters

The OsModel, required by every algorithm, will provide the basic platform dependent code, the radio (used in the communication), the clock, from which the time is acquired and to be set, the debug channel to send debug messages, the graph algorithm which will build the tree, and a maximum number of nodes to set the data structure size. Note that no timer is needed in this algorithm, because its running starts once the tree is built.

As explained before, the graph algorithm is called at the `init()` function. Once it has finished the tree construction, the `start_synchronization()` function is called. This function timestamps and sends a message to each of its neighbors:

```
start_synchronization( )
{
   // Send the synchronization pulse to all children
   synchronizationMessage.set_msg_id( LtsMsgIdSynchronizationPulse );
   for ( int i = 0; i < (int)neighbors_.children_.size(); ++i ) {
      synchronizationMessage.set_t1( clock().time() );
      radio().send( neighbors_.children_[i], 1 + TIME_SIZE,
                    (uint8_t*)&synchronizationMessage );
   }
}
```

**Listing 17: LTS start synchronization** - starts the synchronization, called when the tree is built. Sends a timestamped message to each neighbor.

The rest of the algorithm is implemented in the `receive` function. Its implementation can be consulted in the Program code 40 on page 106.

### 3.4.3 Simulations

In this case, the simulations are not of much use (only useful to test the tree construction part), because the Shawn simulator's nodes have all the same time, so it is not possible to test if the synchronization has taken place, and more importantly what is its performance.

They were used, still, tot test the tree construction and the correct message exchange in the synchronization phase, and performed correctly.

## 3.5 TPSN: Time-sync Protocol for Sensor Networks

The Time-sync Protocol for Sensor Networks algorithm (12) is also based in the sender-receiver schema, and the main difference between TPSN and LTS is the propagation of the synchronization. In (12), it is said to timestamp the messages in the MAC Layer, thus reducing the error caused by message management before sending. This is not possible right now in the WISELIB, because the supported platforms don't implement this functionality.

### 3.5.1 Description of the algorithm

The main difference between TPSN and LTS the way of propagating the synchronization. This algorithm is thought to be more fault tolerant, and it takes care of many common situations with real nodes so problems such as message collision, message loss or different start up time don't affect the running of the algorithm, and every sensor is synchronized.

The algorithm has two phases: **tree construction** and **synchronization**.

#### 3.5.1.1 Tree construction

In this phase of the algorithm, a tree is built having as root the reference node. The tree is built by sending `level_discovery(i)` messages, which have one parameter *i*, that is the level of the node sending the message.

The beginning of the tree construction phase takes place when the reference node sends the message `level_discovery(0)`. Every neighbour that receives a `level_discovery(i)` message, sets its parent in the tree to the sender, and broadcasts a message `level_discovery(i+1)`. Once a parent is taken, this node will stop to listen to `level_discovery` messages.

In the case that a sensor wakes up when the tree is already built (which is detected if no `level_discovery` message was received after a timeout), the sensor will send a `level_request` message. Every neighbour will answer with a `level_discovery(i)` message, and after a timeout, the new coming node will choose as a parent the neighbor with lower level.

#### 3.5.1.2 Synchronization

In this algorithm, even though the synchronization follows the sender-receiver schema as in LTS, it has some differences. Note that the synchronization in LTS was started by the parent, which later calculated the offset, and then notified it to the child. This meant that each pair's synchronization needed three messages.

In TPSN the synchronization is started by the child, and so the synchronization only needs two messages, because the offset is calculated by the child and doesn't need to be notified.

The synchronization begins with the reference node broadcasting a `time_sync` message. After hearing this message, all the level 1 nodes will wait a random time, and then send a `synchronization_pulse` to the reference node, which will answer an `acknowledgement`. The `acknowledgement` contains the four timestamps needed to calculate the offset and adjust the clock.

The `acknowledgement` is sent by broadcast, so when a child of the receiver of this message hears it, it knows that its parent will be synchronized shortly, and so starts its synchronization after waiting a random time. By waiting random times, the probability of message collision is reduced greatly.

If a sensor attempts to synchronize to its parent three times, without receiving an acknowledgement, it considers that its parent is dead, and sends a `level_request` message in order to find a new parent.

This algorithm achieves synchronization of all the connected network using three messages per node (one for the tree construction and two for the synchronization), and considers possible scenarios (message loss, message collision, sensor appearing, sensor disappearing) and provides a way of dealing with them. It is more complete (and complex) than LTS.

### 3.5.2 Implementation

The TPSN algorithm, by having its own method of building the tree and spreading the synchronization, doesn't need the previous work of any other algorithm, not even a neighborhood discovery phase is needed. From what was explained before, we can deduce that the algorithm will make extensive use of the timer, because many actions take place after a random timeout.

Starting with the definition of the class, its template parameters are the ones we can expect from a synchronization algorithm:

```
template<typename OsModel_P,
         typename Radio_P = typename OsModel_P::Radio,
         typename Debug_P = typename OsModel_P::Debug,
         typename Clock_P = typename OsModel_P::Clock,
         uint16_t MAX_NODES = 32>
class TpsnSynchronization
```

**Listing 18: TPSN template parameters** - operating system, radio, debug channel, clock and maximum number of nodes are passed as template parameters.

It has the operating system model, the radio, debug channel, clock and maximum number of nodes. If the default values are used, only the operating system model is needed to get an instance of this algorithm.

The TPSN algorithm has a message class, such as LTS did, used to send and obtain the data exchanged between the sensors. It contains a buffer:

```
block_data_t buffer[1 + TIME_SIZE*3 + NODE_ID_SIZE];
```

**Listing 19: TPSN message buffer** - buffer used to send TPSN messages, containing id, timestamps and node ids.

The maximum size is this because we need one byte as the message id, three timestamps in the acknowledgement message, and also the receiver id must be sent (because the messages are broadcasted, in order to notify other neighbors to start synchronization, as explained in the previous section). The `TpsnSynchronizationMessage` class also includes functions to set and get the data stored in the buffer. The message ids needed by this algorithm are:

```
enum TpsnSynchronizationMsgIds {
   TpsnMsgIdLevelDiscovery = 230, ///<  LEVEL_DISCOVERY
   TpsnMsgIdLevelRequest = 231, ///<  LEVEL_REQUEST
   TpsnMsgIdTimeSync = 232, ///<  TIME_SYNC
   TpsnMsgIdSynchronizationPulse = 233, ///< SYNCHRONIZATION_PULSE
```

```
   TpsnMsgIdAcknowledgement = 234, ///<  ACKNOWLEDGEMENT
};
```

**Listing 20: TPSN message ids** - message ids for `LevelDiscovery`, `LevelRequest`, `TimeSYnc`, `SynchronizationPulse` and `Acknowledgement`

No special data structure is needed, as the tree is stored by the children knowing who is its parent. But it is important to note the extensive use of the timer in this algorithm, which depends on many timeouts, and so the following variables are used:

```
   millis_t root_startup_time_, tree_construction_time_,
            random_interval_time_, timeout_;
```

**Listing 21: TPSN variables** - variables used by the algorithm.

- `root_startup_time`: milliseconds the root waits before starting the tree construction.
- `tree_construction_time`: milliseconds the root waits between starting the tree construction and starting the synchronization.
- `random_interval_time`: the time for starting the synchronization will be a random value between zero and this time.
- `timeout`: after waiting `timeout` milliseconds, a sensor waiting for an answer will consider its previous message lost. After three consecutive timeouts, the receiver will be considered dead.

At the beginning of the execution, the `enable()` function is called:

```
enable( void )
{
   levelDiscoveryMessage[0] = TpsnMsgIdLevelDiscovery;
   if ( level_ == 0 )
   {
      built_tree_ = false;
      timer().template set_timer<self_type,
                        &self_type::timer_elapsed>(
                        root_startup_time_, this, 0 );
```

```
    }
    else
    {
        synchronized_ = false;
        retries_ = 0;
        new_level_ = -1;
        timer().template set_timer<self_type,
                        &self_type::timer_elapsed>(
                        timeout_, this, 0 );
    }
}
```

**Listing 22: TPSN enable** - enable function, starts the algorithm. Starts the tree construction if the node is root, and sets a timeout if not. After the timeout the node will request a parent if it doesn't have one

The `enable()` function distinguishes two cases: if the node is a reference node or not. In each case it does:

1. **root**: waits `root_startup_time`, after which the tree construction will start.
2. **normal node**: waits `timeout`. After that, it will request a father if it still doesn't have one.

This algorithm has two main functions which control all its behaviour: `receive()` and `timer_elapsed()`. The `receive()` function identifies the messages received, and calls the `timer_elapsed()`, as every operation in this algorithm is executed after a certain timeout. The implementation of the `timer_elapsed()`, can be consulted in Program code 41 on page 41.

### 3.5.3 Simulations

This algorithm was also simulated using the Shawn simulator, with the aim of testing that the tree is built correctly and the synchronization is spread as expected. Again, just as in LTS, it is not possible to test in the simulator if the synchronization takes place, but this will be tested in a real world scenario.

The simulator also offers the possibility to test if the solutions for the special cases work, by setting a probability of message loss, and also by turning on and off nodes during the experiment.

## 3.6 HRTS: Hierarchy Reference Time Synchronization

The Hierarchy Reference Time Synchronization algorithm (10), being a receiver-receiver algorithm, achieves the synchronization of nodes in a different way as the two algorithms explained before. In a receiver-receiver algorithm, the timestamps compared aren't from the reference node and the receiver, but from two different receivers instead.

### 3.6.1 Description of the algorithm

In an scenario with a Base Station (BS, reference node), and 5 other sensors, numbered from $n_1 \ldots n_5$. The algorithm would work as follows:

1. BS timestamps a message with $t_1$, picks up a receiver (e.g. $n_2$), and broadcasts a message containing both data. All interested nodes record timestamp $t_1$.

2. Node $n_2$ replies the message, sending the timestamp $t_2$ (at which it received the message), and $t_3$, at which it sent the reply.

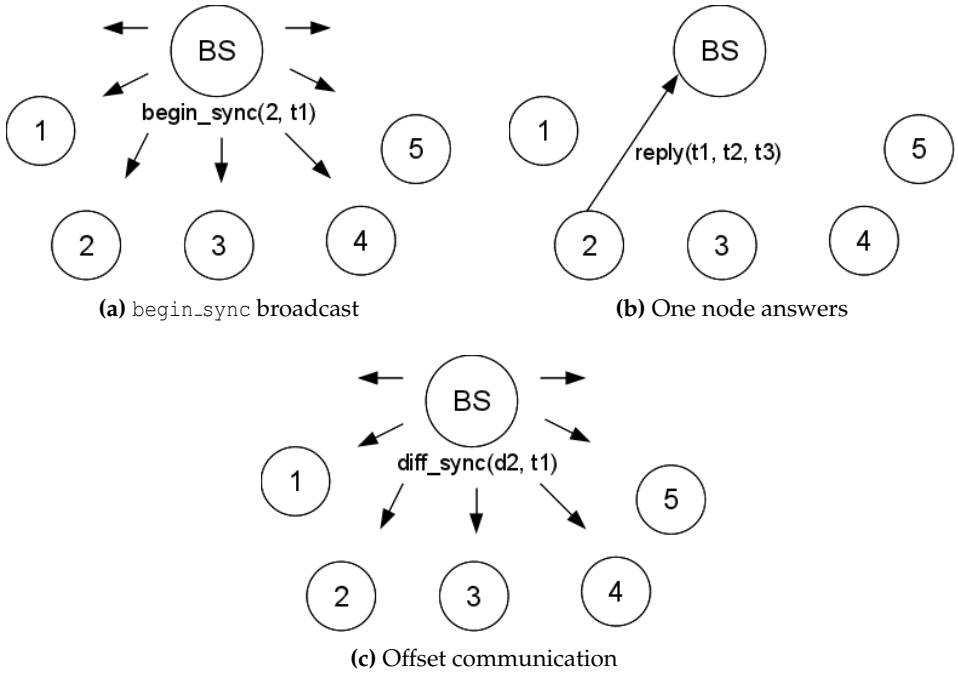
3. BS records $t_4$ at which it received the message from $n_2$. Knowing all four timestamps, it calculates n2's offset as $d_2 = \frac{t_1+t_4-t_2-t_3}{2}$. The BS broadcasts then $t_2$ and $d_2$.

4. All interested nodes will synchronize knowing this data. For example $n_3$, which received the first broadcast at time $t'_2$, calculates its offset $d'$ as:

$$d' = d_2 + t_2 - t'_2$$

$$T = t + d'$$

5. Sensors which synchronized in the previous step, will now start as base stations from step 1, thus spreading the synchronization.




**(a)** `begin_sync` broadcast **(b)** One node answers

**(c)** Offset communication

**Figure 3.10:** Execution in parallel of DBFS. After a Label and an Echo phase, one level of the tree is built.

In Figure 3.10 we can see the three steps in the receiver-receiver synchronization: first a message is broadcasted and timestamped by everyone. Then one node answers, and the Base Stations calculates the offset. Finally, he Base Station broadcasts the offset, and the other nodes adjust their clocks as explained in section 3.2.2.

Once a sensor is synchronized, it broadcasts a `sync_begin` message and starts the synchronization to all its neighbors that have not yet been synchronized.

### 3.6.2 Implementation

This algorithm's implementation differs from the ones explained before in two main functions: `start_propagation()` and `receive()`. These two functions control all the behaviour of the algorithm.

The first one, `start_propagation()`, timestamps and sends the first synchronization message:

```
start_propagation( )
{
   if (neighborhood_.neighbors_.empty())
      propagated_ = true;
   else {
      message.set_msg_id(HrtsMsgIdBeginSync);
      message.set_receiver(neighborhood_.neighbors_[0]);
      t1 = clock().time();
      message.set_t1( t1 );
      radio().send( radio().BROADCAST_ADDRESS,
                    sizeof( SynchronizationMessage ),
                    (uint8_t*)&message );
   }
}
```

**Listing 23: HRTS start propagation** - starts synchronization with itself as base station.

The `receive()` function, which can be found in the Program code 42 on page 109 implements the rest of the functionality.

### 3.6.3 Simulations

Through simulation with Shawn, it was possible to determine that the synchronization propagates through the network. It is necessary now to test the synchronization error on real nodes, which is explained in the following chapter.

## 3.7 Synchronization tests with WISELIB on iSense

### 3.7.1 Experiments discussion

In this section it is explained how the synchronization was tested using real iSense sensors from UPC.

The first challenge we had to face is how to measure the dis-synchronization of sensors. In a first thought, we can think of using the debug attached to a PC, but this uses the serial port, which only gives us a precision of milliseconds, and we are trying to achieve better results.

Another possibility was to use radio messages sent at certain times, but this still isn't precise enough. If our method's error is determined by some uncertainties in radio message sending, it is obvious that this is not to be used to calculate the clock's offset.

### 3.7.2 Description of the tests

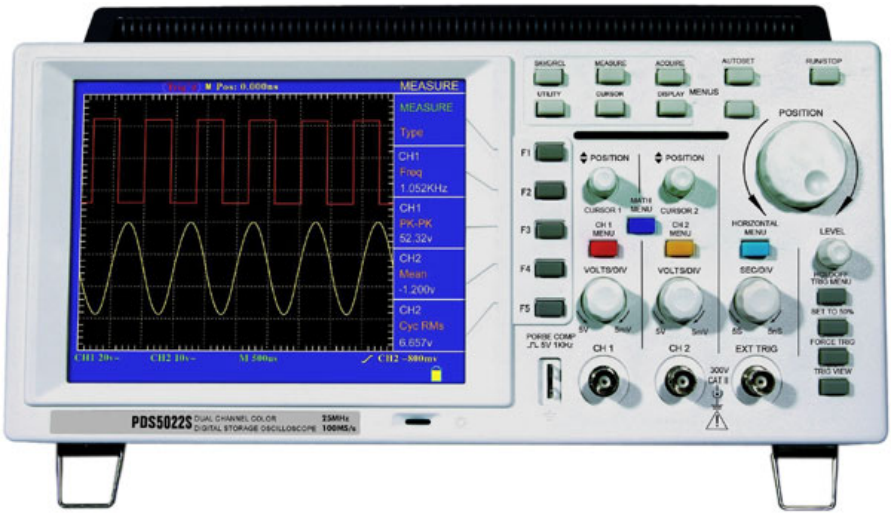



**Figure 3.11: Owon oscilloscope** - Owon oscilloscope used for clock offset determination

Finally, we decided to use an oscilloscope which will give us as mush precision as we deserve (down to nanoseconds). The oscilloscope used, which can be seen in Figure 3.11, lets us use two input channels. Each of them is attached to a LED (Light Emision Diode), which will turn on for ten milliseconds at certain times.

The oscilloscope lets us see the input of both channels on the screen, and when the sensors are turned on, we can see that they are not synchronized. For this we set a trigger, which will freeze the image every time the first led is turned on. This way we have a frozen image of how dis-synchronized both sensors are, and adjusting the time scale in the oscilloscope we can calculate the amount of the offset.

At this moment, the algorithm is run, and after some messages are exchanged, the clock will be adjusted. We will be able to see how both LEDs blink at the same time, and with the oscilloscope we will be able to determine what is the offset.

For this experiment to work, both of the sensors need to be one next to the other, because we need both of the to be connected to the oscilloscope. In order to test that

the synchronization takes places also on sensors at many hops, we used an application which had the edges of the network hardcoded, making sensors which were close physically, to be at many hops distance.

### 3.7.3 Results

As it is obvious from the source of the error in synchronization algorithm, the error is added at each hop from the reference node to the leaves of the tree. It is for this reason that the most interesting error to calculate is the error in a pairwise synchronization.

The test described earlier was done on two iSense nodes connected to an oscilloscope, and the results are on Figure 3.12. The results have the expected behaviour: they should present a form of a a Gaussian distribution centered on zero, and our results have more or less this form. Note that we added together the positive and negative errors, and in our representation in Fig. 3.12 we only see half of the Gaussian distribution.

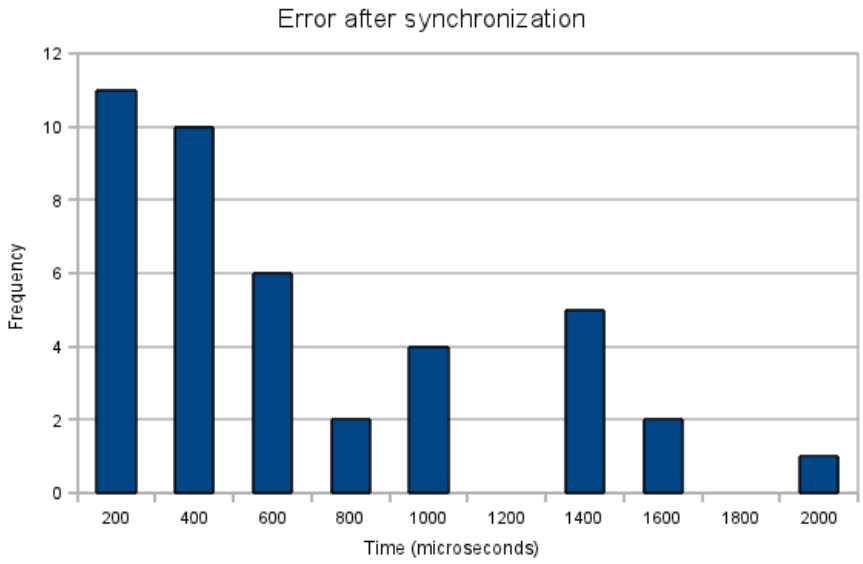



**Figure 3.12: Pairwise synchronization error** - histogram of the error in the synchronization of two iSense sensors

About the magnitude of the error, we can see that we usually get an error of less than one millisecond. This is very easily explained because of the precision of the iSense clock, which gives the time with an accuracy of milliseconds. If better results were

needed, it would be necessary to implement a more precise clock, which would make possible to reduce the error to microseconds.

During the tests it was made obvious that the clocks are not very precise, and that they tend to dis-synchronize all the time in a linear fashion. It would be provably very useful to take this clock drift into account, not only for achieving better synchronization but to maintain in through time.

## 3.8 Algorithms comparison

The three algorithms to compare are LTS, TPSN and HRTS. Let's first have a look at Table 3.1, an then comment its content.

| | LTS | TPSN | HRTS |
|---|---|---|---|
| **Algorithm type** | sender-receiver | sender-receiver | receiver-receiver |
| **Number of messages** | 3·nodes+tree construction | 3·nodes | 3·(broadcast domains) |
| **Error** | $\tau + \eta$ | $\tau + \eta$ | $\tau$ |
| **Fault tolerance** | No | Yes | No |

**Table 3.1:** Synchronization algorithms comparison

From this table we can note the superiority of TPSN with respect to LTS. They both are sender-receiver algorithms and thus have the same error of synchronization (see the $\tau, \eta$ definitions in sections 3.2.1 and 3.2.2). TPSN can build the tree and synchronize the whole network using three messages per node; LTS, on the other hand, requires three messages for the synchronization, and needs the tree built beforehand. Even more, TPSN includes fault tolerance: can fix special cases such as a node appearing or disappearing, message loss... For all these reasons we can conclude that TPSN is a better algorithm than LTS.

HRTS algorithm, for being a receiver-receiver algorithm, has less synchronization error than TPSN. It also uses less messages (in the worst case, being a linear network it would use the same). Messages in HRTS are broadcasted, so only three messages are needed per broadcast domain, thus having much less messages in dense networks.

However, HRTS doesn't contemplate common errors such as package loss, which can imply poor results in real world applications.

In conclusion, if good synchronization is desired, the best choice is to use HRTS, and if the algorithm will be run in a very changing network, with obstacles or interferences, the best algorithm to use is TPSN.

# Chapter 4

# Topology

## 4.1 Topology algorithms

Given a Wireless Sensor Network, working at full power, we can model it as a graph $G(V,E)$. The sensors are the nodes and the edges join sensors which can communicate at full power. This graph is typically very connected, so it would be possible to reduce the power of many sensors' radio, and still preserve desirable properties such as connectivity and link bi-directionality.

Topology algorithms build a logical subgraph in this maximum power graph, thus allowing sensors to reduce radio power and range, as we can see inf Figure 4.1. These algorithms can let other algorithms work on top of them more efficiently, by sending messages to less sensors. If messages are sent to less sensors, specially if they are sent to the nearest ones, three main goals are accomplished:

- **Energy preservation**: messages are sent to the closest neighbors, so they can be sent using less power. Moreover, if messages are only sent to the closest neighbors, less messages are sent and more energy is saved.
- **Less collisions**: less messages are sent, and using less power. This reduces the probability of two messages for the same node arriving at the same time, then producing a message collision.
- **Less redundant communications**: reducing the graph connectivity, we help other algorithms working on top of the topology, by reducing the possible paths from one node to another, which can lead to redundant communication.

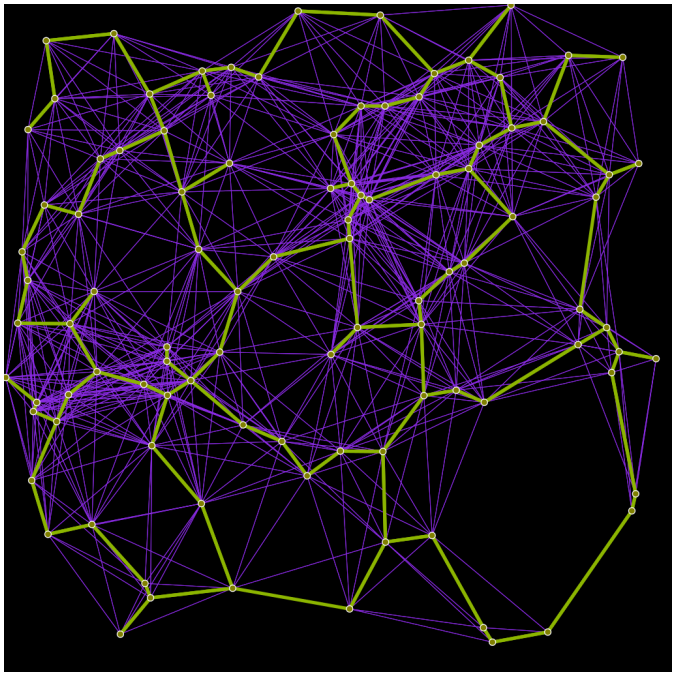


**Figure 4.1: LMST 25 nodes** - LMST topology (yellow links) over the maximum range graph (purple links)

We will name *topology* algorithms those algorithms whose result is a subgraph of the network, and *topology control* algorithms, those algorithms that reduce the radio power, and thus physically create the subgraph. In general it was chosen to implement the algorithms as topology algorithms, in order to let the user choose to apply or not the radio range reduction.

In this chapter we will see some examples of topology algorithms. We will explain the algorithms, their implementation into the Wiselib library, and the tests that were run in order to test their correctness and to compare them. The algorithms commented, with more or less detail, are LMST (20), FLSS (19), KNEIGH (5), XTC (31) and CBTC (18). For further information on other topology control algorithms, refer to (26) or (25).

## 4.2 LMST: Local Minimum Spanning Tree

It is widely discussed the need of topology control algorithms that build topologies where the main goal is to reduce energy consumption in sensors communication while preserving connectivity, preferably bidirectional. This algorithm builds an approxima-

tion of a minimum spanning tree, which is a good approach for getting minimal energy consumption.

Local Minimum Spanning Tree (20) is a distributed algorithm, in which every sensor in the network generates locally a minimum spanning tree of its visible neighborhood and then chooses its closest nodes as its topology.

### 4.2.1 Description of the algorithm

The algorithm is executed periodically, and has three main phases:

1. **Neighborhood discovery**: every sensor broadcasts its position, and stores the ones it receives.

2. **Topology construction**: knowing the position of its neighbors and itself, it builds a minimum spanning tree using Prim's algorithm (22). Its topology neighbors are the ones that are at one hop distance in this minimum spanning tree.

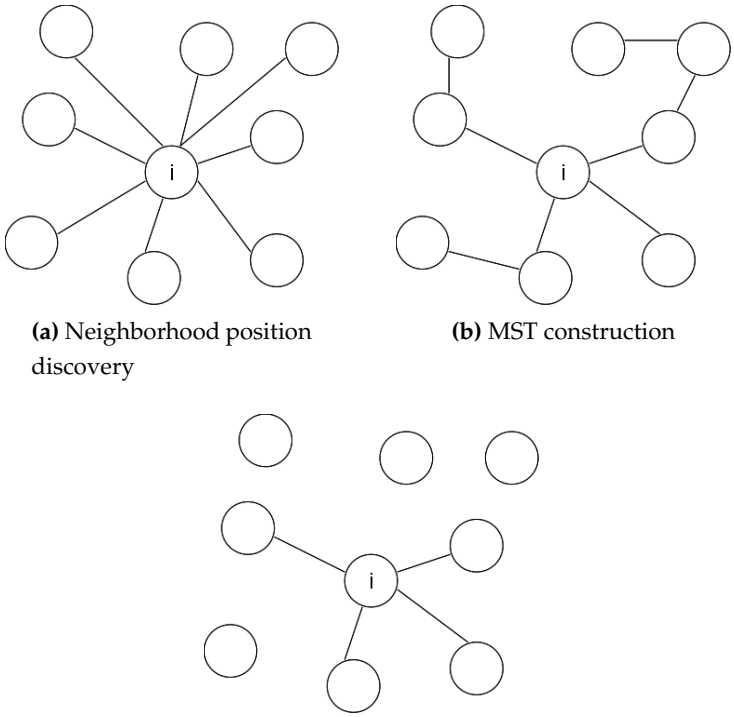
3. **Link bi-directionality**: links might be unidirectional, as explained in (20), and we convert them all to bidirectional ones (instead of deleting the unidirectional), the reasons will be explained in the next section.

We can see the two first phases in Figure 4.2. First, position information is received from the neighborhood. Next, Prim's algorithm is run, and the minimum spanning tree is found. The resulting topology is formed by one-hop nodes in this tree.

#### 4.2.1.1 $G^+$ choice

The original algorithm gives two options for the implementation, given the graph $G$ generated in step two, you can convert it to $G^+$, making unidirectional edges bidirectional, or removing them, thus getting $G^-$. We decided to use $G^+$ for the following reasons:

1. A first consideration is the number of messages being exchanged to add or delete an unidirectional link. To make it bidirectional, one message is needed: each node sends to its neighbors `You are my neighbor`, and the receiver adds it as a neighbor if it doesn't have it.




**(a)** Neighborhood position discovery

**(b)** MST construction

**(c)** Topology: on-hop neighbors in MST

**Figure 4.2:** Execution of LMST by node i.

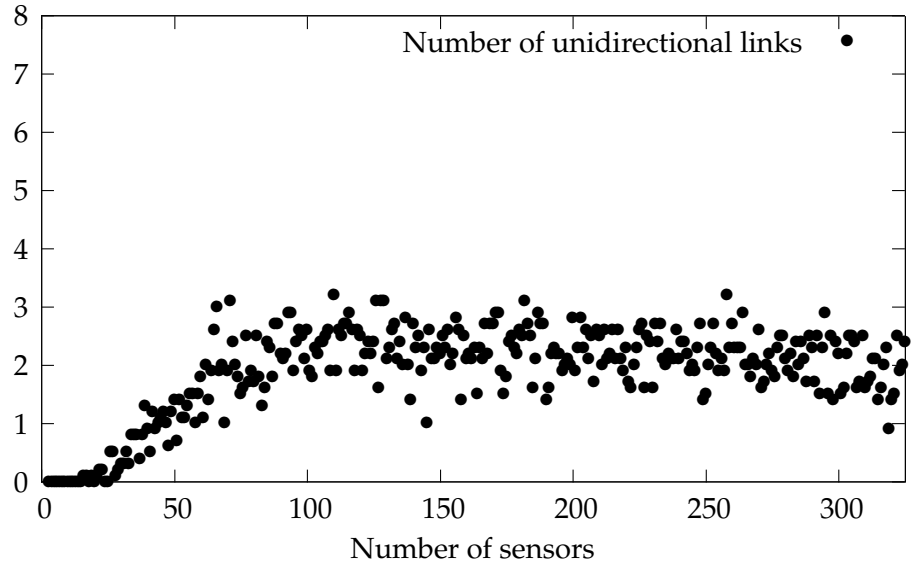



**Figure 4.3: LMST unidirectional edges** - Each point is the mean of 10 simulations in which sensors were placed in a rectangular 100x100 grid with a radio range of 25.0.

If we wanted to delete unidirectional links, two messages would be needed: `Am I your neighbor?`, which should be answered with `Yes` or `No` so the link is removed or not.

The second approach would be chosen if unidirectional links were very common, in which case sending two messages would be useful for getting a minimal graph, but through simulation we discovered this case isn't very usual, as shown in Figure 4.3. In our simulations, we placed randomly different number of sensors in a rectangular grid, and the result is that the number of unidirectional links is small in comparison to the number of sensors, and remains constant.

2. While both $G^+$ and $G^-$ maintain connectivity in an obstacle-free scenario, only $G^+$ does in some situations with obstacles, for example in Figure 4.4. In this example, when having a wall, sensor 3 deduces sensor 1 and 2 will be connected (but aren't), and sensor 1 knows it has to connect to 3. Therefore, the only way to preserve connectivity is to convert unidirectional links to bidirectional.

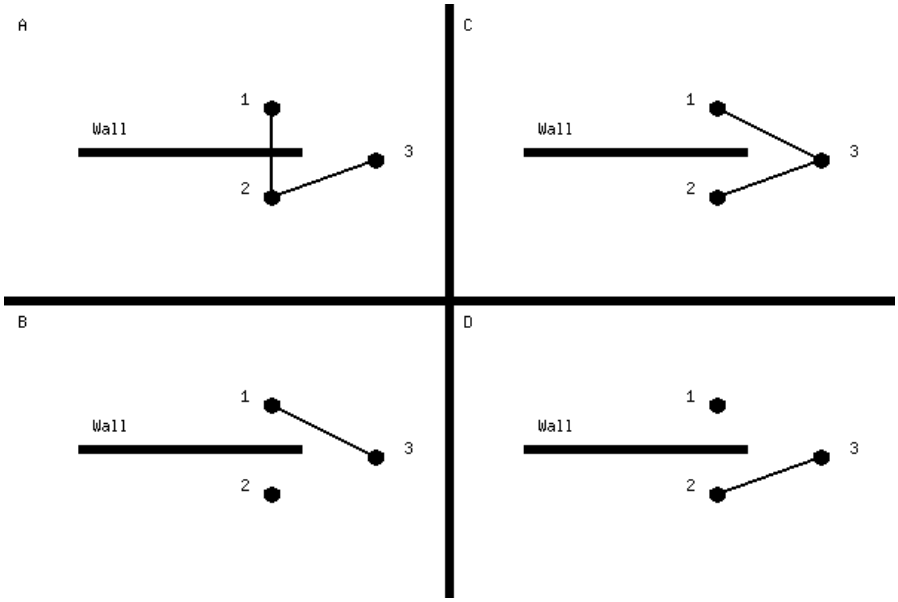



**Figure 4.4: LMST scenario with walls** - A: point of view of 3; B: point of view of 1; C: $G^+$: connected; D: $G^-$: unconnected

### 4.2.2 Implementation

The implementation of LMST starts with its header:

```
template<typename OsModel_P,
   typename Localization_P,
   typename Float=double,
   uint16_t MAX_NODES = 32,
       typename Radio_P = typename OsModel_P::Radio,
       typename Timer_P = typename OsModel_P::Timer>
class LmstTopology :
   public TopologyBase<OsModel_P>
```

**Listing 24: LMST header** - Template parameters: operating system, localization method, floating point number, maximum number of nodes, radio and timer.

We must note the parameter `Localization_P`, which is the one that will provide the position of the sensor, needed by LMST. This positioning method can be implemented by any class with the specified functions, and this allows the application to use GPS, hardcoded positions, a localization algorithm...

The data structures used for the implementation of LMST are:

- Static vectors for storing the Visible Neighborhood (ids and positions), for the topology itself, and for Prim's algorithm implementation: distance from the root to a sensor, each sensors' father, and a vector of booleans which indicates if a node is in the priority queue.
- A max heap priority queue used in Prim's algorithm.

These vectors are declared as follows:

```
vector_static<OsModel, node_id_t, MAX_NODES> NV;
vector_static<OsModel, Position, MAX_NODES> Pos;
vector_static<OsModel, position_t, MAX_NODES> D;
vector_static<OsModel, node_id_t, MAX_NODES> p;
vector_static<OsModel, bool, MAX_NODES> is_in_PQ;
priority_queue< OsModel, PPI, MAX_NODES*10 > PQ;
```

**Listing 25: LMST data structures** - data structures used to represent the neighborhood, the topology, and needed by Prim's algorithm.

The most important function in this algorithm is the `generate_ topology` function, which implements Prim's algorithm and selects the 1-hop sensors which will form the topology. The implementation of this function can be found in 43, on page 111.

### 4.2.3 Simulations

To test the correctness of the LMST algorithm, many tests were done using the Shawn simulator (simulations are explained in detail in Appendix B. It is very useful to visualize the resulting topology, in order to find anomalies which can imply bugs in the implementation. In figure 4.5, 4.6 and 4.7 we can see the LMST topology in scenarios with 100, 1000 sensors and 1000 sensors with more range.

Another test, which is more subtle than the visualization of the topology is to compute the average degree of the nodes in the resulting topology. (20) stated the average degree for simulations with some parameters. Our simulations resulted with:

And as $2.06 \simeq 2.05628$, we can confirm that the results in (20) are correct and the same as ours.

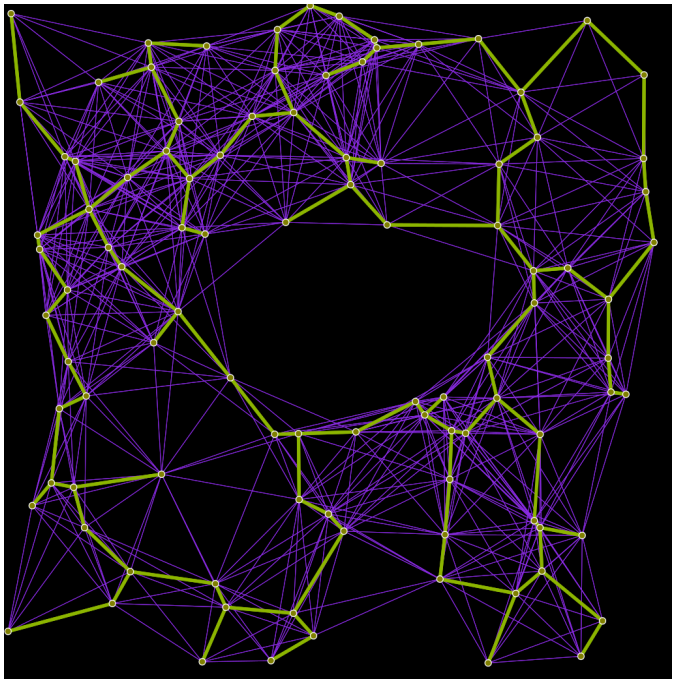


**Figure 4.5: LMST 100 nodes** - simulation of 100 nodes and a range of 25 in a 100x100 scenario

| | Nodes | Range | Degree |
|---|---|---|---|
| **(20) results** | 100 | 25.0 | 2.06 |
| **Simulations** | 100 | 25.0 | 2.05628 |

**Table 4.1: LMST theoretical and simulation results** - comparison of theoretical results in (20) and simulations in Shawn.

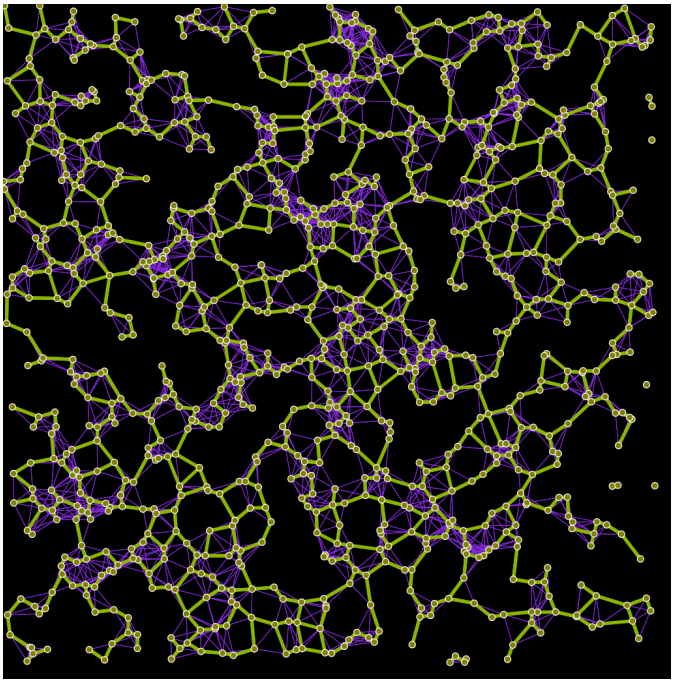


**Figure 4.6: LMST 1000 nodes** - simulation of 100 nodes and a range of 5 in a 100x100 scenario

### 4.2.4 Tests on UPC testbed

In Figure 4.8 we can see the resulting topology after running the LMST algorithm in the UPC testbed, located in the 2n floor of building Omega.

## 4.3 FLSS: Fault-Tolerant Spanning Subgraph

The Fault-Tolerant Spanning Subgraph algorithm (19), while trying to reduce the energy consumed by the network, takes into account that wireless communications often have problems (message collision, interference, obstacles, changing network), which can lead to temporal or permanent message loss. Having this in mind, it doesn't seem desirable to have a minimal topology if we want to assure connectivity.

The FLSS algorithm will use the concept of $k$-vertex connectivity, i.e. $k-1$ vertexes and all their edges can be removed preserving the connectivity of the network. FLSS wild build a $k$-vertex connected network, with values of $k$ of 2 or 3, thus having fault tolerance in case of node failure.

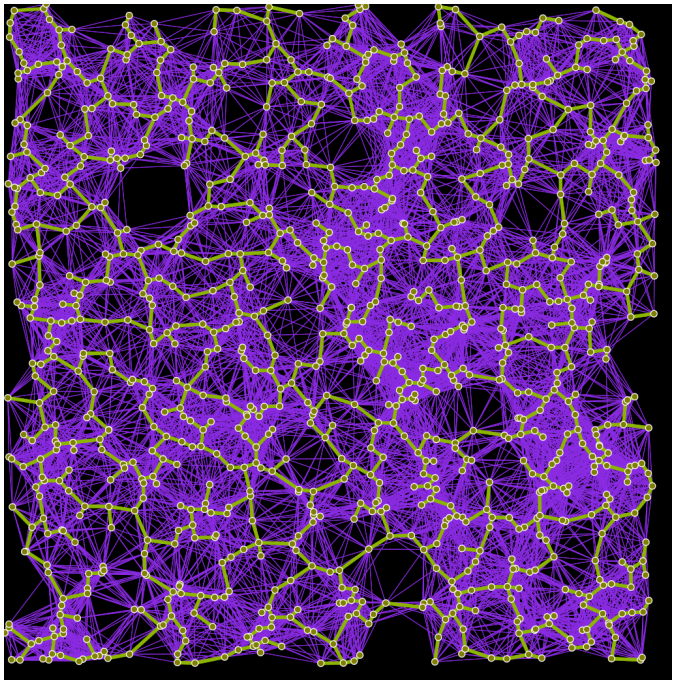


**Figure 4.7: LMST 1000 nodes, more range** - simulation of 100 nodes and a range of 10 in a 100x100 scenario

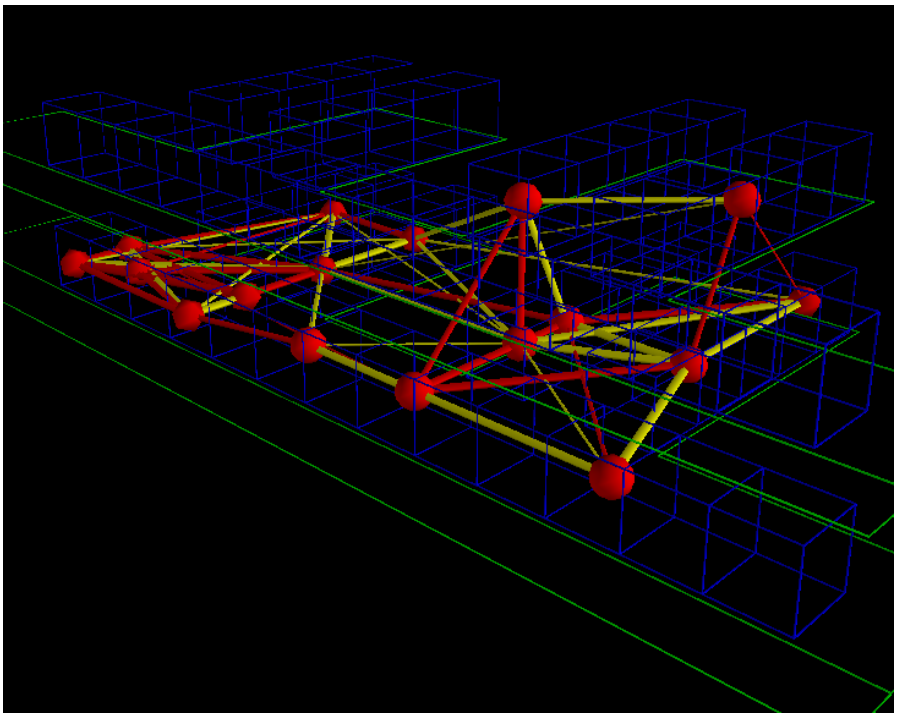


**Figure 4.8: LMST in UPC testbed** - LMST results in UPC testbed

### 4.3.1 Description of the algorithm

The FLSS algorithm is very similar to LMST, as its running phases are the same:

1. **Neighborhood discovery**: collect the position data of all neighbors.
2. **Topology construction**: select a subset of nodes which will form the topology.
3. **Link bi-directionality**: convert unidirectional links to bidirectional ones.

The difference between LMST and FLSS is that in phase 2 a different algorithm is used. Prim's algorithm used in LMST only assures 1-connectivity, and in FLSS we need $k$-vertex connectivity. The algorithm used will be FGSS$_k$: Fault-tolerant Global Spanning Subgraph.

#### 4.3.1.1 FGSS$_k$ algorithm

FGSS$_k$ algorithm is a generalization of Kruskal's algorithm (15). FGSS$_k$ starts with an empty set of vertexes and edges, and adds edges (and their nodes) in ascending order of edge length, until all vertexes are in the same $k$-connected component. A pseudocode for this function is in Figure 4.9.

To check if node $u_0$ is $k$-connected to node $v_0$, network flow techniques are used (11), which consist of calculation the maximum flow between $u_0$ and $v_0$, given a binary flow of $0/1$ to each edge. If the maximum flow is greater or equals $k$, the vertexes are $k$-connected.

#### 4.3.1.2 FLSS$_k$ algorithm

FLSS$_k$ is a localized algorithm, in which each sensor calculates its FGSS$_k$ with one modification: the algorithm stops when the node running the algorithm is $k$-connected to all its neighbors. Once the spanning subgraph is obtained, the topology derived is the one consisting of 1-hop neighbors in this spanning subgraph.

### 4.3.2 Implementation

This algorithm has the same requirements as LMST: position knowledge, so its template parameters are exactly the same.

**Procedure:** FGSS$_k$
**Input:** $G(E,V)$, a $k$-connected simple graph;
**Output:** $G(E_k,V_k)$, a $k$-connected spanning subgraph of $G$;
**begin**
  $V_k := V$, $E_k := \emptyset$;
  Sort all edges in $E$ in ascending order of length;
  **for** each edge $(u_0,v_0)$ in the order
    **if** $u_0$ is not $k$-connected to $v_0$
      $E_k := E_k \cup (u_0,v_0)$;
    **else if**
      exit;
    **endif**
  **end**
**end**

**Figure 4.9: FGSS$_k$ pseudocode** - pseudocode for generalized Kruskal algorithm.

The implementation of the algorithm is done using the pseudocode in 4.9, and implementing the Ford-Fulkerson maximum flow algorithm for the $k$-connectivity test. The data structures needed by this algorithm are:

```
typedef pair<int, int> PII;
vector_static<OsModel, node_id_t, MAX_NODES> NV;
vector_static<OsModel, Position, MAX_NODES> Pos;
vector_static<OsModel, vector_static<OsModel, uint8_t, MAX_NODES>, MAX_NODES> capacity;
vector_static<OsModel, vector_static<OsModel, node_id_t, MAX_NODES>, MAX_NODES> G;
priority_queue< OsModel, pair<float_t, PII >, MAX_NODES*10 > E;
vector_static<OsModel, bool, MAX_NODES> seen;
vector_static<OsModel, node_id_t, MAX_NODES> from;
std::queue<node_id_t> Q;
```

**Listing 26: FLSS data structures** - data structures used to represent the neighborhood and needed by the Ford-Fulkerson algorithm.

The only difference between LMST and FLSS in the implementation is the function `generate_topology()`, because the neighborhood discovery and link bi-directionality phases are the same. The implementation of the new `generate_topology()` function,

including the `max_flow()` routine can be found in Program code 44 on page 114.

## 4.4 Other topology algorithms

In this section, other topology algorithms will be explained more briefly. They were implemented by some partners of the project in UPC and I helped in their implementation occasionally. It will give us a broader look at topology algorithms. These algorithms are: CBTC (implemented by Josep Anguera) and KNEIGH and XTC (implemented by Juan Farré).

### 4.4.1 CBTC: Cone-Based Topology Control

#### 4.4.1.1 Description of the algorithm

The CBTC (18), (21), divides the space in cones of width $\rho$, and picks up the closest neighbor contained in each cone. Then sets the radio power to a minimum in order to reach that sensor. This algorithm requires a directional antenna, which none of the other algorithm described requires. This antenna will let the algorithm determine from which direction the messages are received, and also to adjust the power for each direction.

### 4.4.2 KNEIGH: K-Neighbors algorithm

#### 4.4.2.1 Description of the algorithm

The KNEIGH (5) algorithm chooses for every node the $K$ closest neighbors, based on distance estimates. This builds a directed graph, from which unidirectional edges will be removed. The resulting subgraph is the final topology.

To estimate the distance, the Wiselib offers the possibility of having a method as a template parameter. This way, exact distances can be given to static nodes via hard-coded data, or distances can be calculated using a positioning module.

#### 4.4.2.2 Tests on UPC testbed

In the figures 4.10, 4.11 and 4.12 we can see the results of running the KNEIGH algorithm on the testbed deployed in Omega building in UPC. With K=3, we see the topology doesn't achieve connectivity. With K=4 the topology is connected, and with topology K=6 we can see the network is very connected.

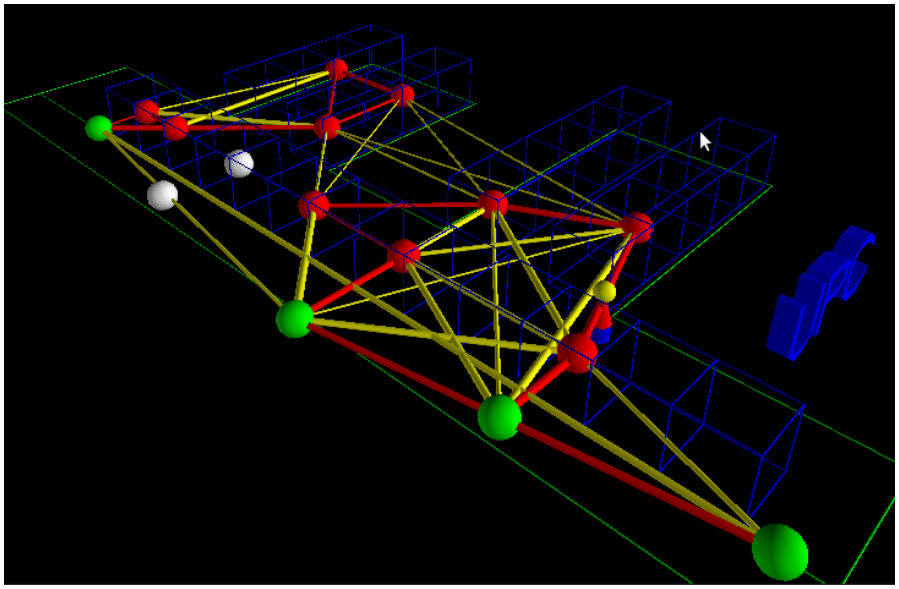


**Figure 4.10: KNEIGH with K=3 in UPC testbed** - KNEIGH results in UPC testbed

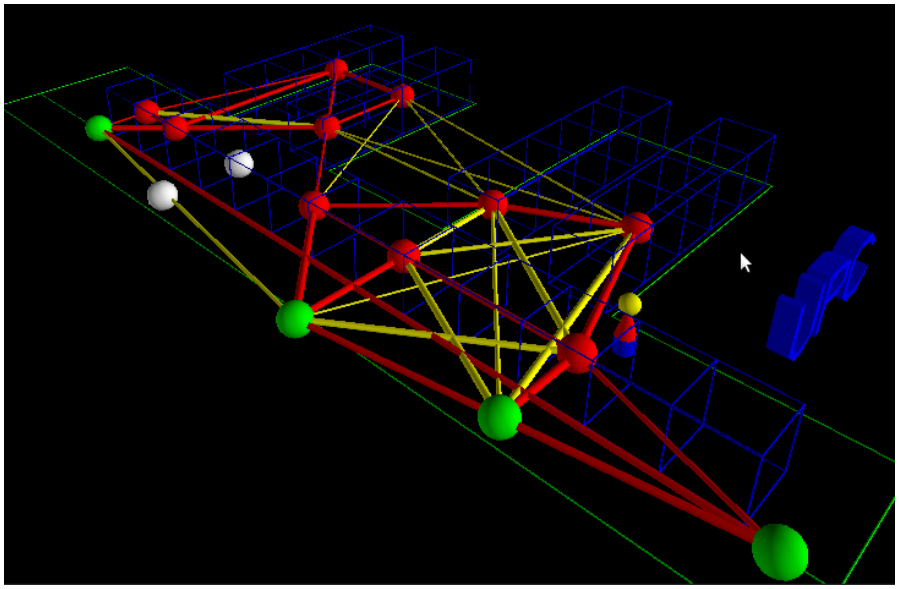


**Figure 4.11: KNEIGH with K=4 in UPC testbed** - KNEIGH results in UPC testbed

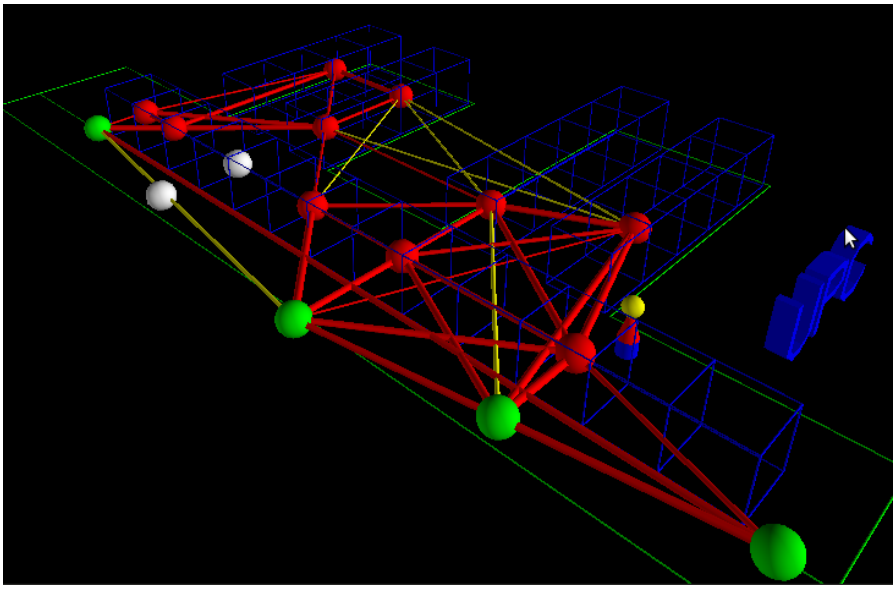


**Figure 4.12: KNEIGH with K=6 in UPC testbed** - KNEIGH results in UPC testbed

### 4.4.3 XTC: X topology control algorithm

#### 4.4.3.1 Description of the algorithm

The XTC (31) protocol, very similar to KNEIGH algorithm differs in two ways:

- Instead of using estimated distances to measure proximity, the concept of link quality is used. The link quality uses the concept of the `ExtendedRadio` in the Wiselib, to get the message quality in addition to the message content in radio communications.
- Unidirectional links are preserved, instead of being deleted.

#### 4.4.3.2 Tests on UPC testbed

On figure 4.13 we can see the result of executing the XTC protocol in our testbed.

## 4.5 Algorithms comparison

In Table 4.2 we see the different algorithms requirements to collect information about their neighborhood, and the method applied to generate the topology.

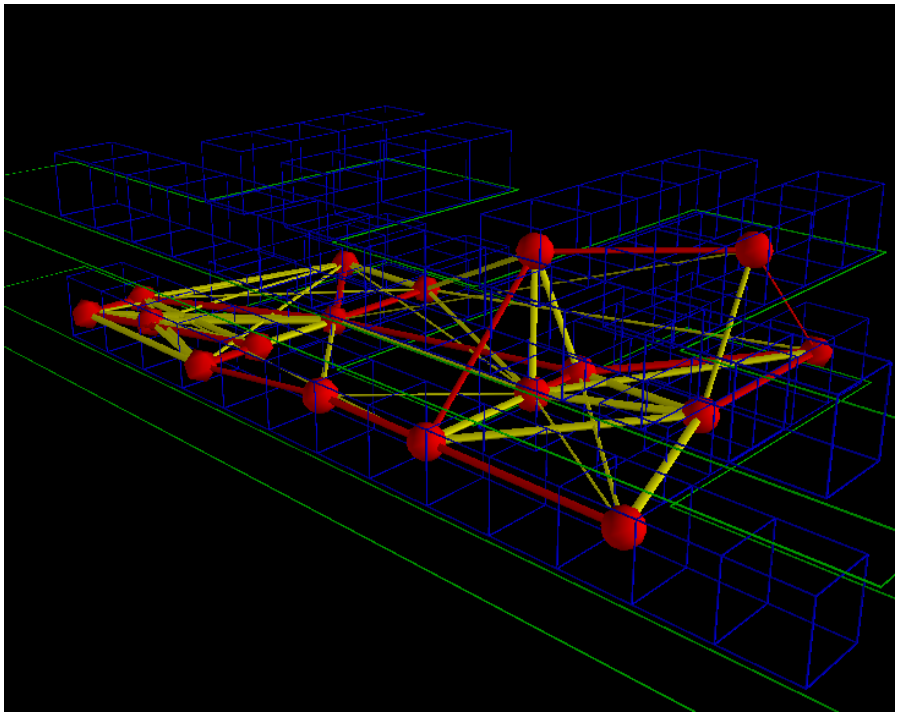


**Figure 4.13: XTC protocol in UPC testbed** - XTC results in UPC testbed

| | **Neighborhood information** | **Topology construction** |
|---|---|---|
| LMST | position | Prim's algorithm |
| FLSS | position | generalized Kruskal's algorithm |
| CBTC | direction & distance | closest neighbors |
| KNEIGH | distance | closest neighbors |
| XTC | link quality | closest neighbors |

**Table 4.2:** Topology algorithms comparison

If looking for a minimal topology, the best choice is LMST, as it builds a good approximation of a minimum spanning tree, and if more connectivity is needed in ordert to introduce fault tolerance, the best algorithm is FLSS, as it offer the minimum energy topology. Both will be only used if the sensors have some way of learning about their position.

But in the case of very limited resorces, or without position information, it is better to use KNEIGH or XTC, which don't assure a minimum energy topology, but behave very well on average. Their requirements are quite low and the algorithm used is very simple and light on the processor.

Finally, the CTBC algorithm also generates a simple topology, using an easy algorithm, but it needs a directional antenna, which is a requirement that we could'nt fulfill on iSense sensors.

# Chapter 5

# Economic analysis

This project, which only has research goals, doesn't have a concrete economic impact. The Wiselib library has been designed having wireless sensor networks' research in mind, but also its later use by companies. The Wiselib (and thus all the algorithms implemented during this project) is public, and licensed under the Lesser General Public License v3.0 (17). This license allows all the code in the library to be included in any application, including closed source ones. This was decided having in mind the use of the library by companies who commercialize sensor network solutions.

It is far beyond the aims of this project to analyze the possible impact on the economy of sensor networks, specially of the algorithms here described. It is interesting, although, to enumerate the expenses that were made to make this project possible. The three main expenses in this project were the hardware: sensors and modules (sensors and modules: GPS, gateway, solar...), the salary earned by grant holders of the project, and travelling and accommodation expenses for the different meetings held during the three years of Wisebed project.

## 5.1 Wisebed finances

The European Union contribution for this three-year project was of 1,477,275.00 €. This is not a detailed financial report on the Wisebed project, just a brief overview on how the EU contribution has been distributed to the partners is presented in Table 5.1.

| | |
|---|---|
| EU prefinancing | 1,477,275.00 € |
| | |
| ITM | 251,871.90 € |
| FUB | 150,016.50 € |
| TUBS | 150,122.50 € |
| RACTI | 150,122.50 € |
| UPC | 150,832.17 € |
| UBERN | 152,961.18 € |
| UNIGE | 150,122.50 € |
| TUD | 150,122.50 € |
| ULANC | 161,870.48 € |
| | |
| Remaining | 9.232,77 € |

**Table 5.1:** Distribution of the first EU contribution

## 5.2 Expenses generated by my work

I will give more detail on the money invested in my work, which has two concepts: salary and travel and accommodation expenses for the meetings. In Table 5.2 are listed these expenses.

| **Salary** | |
|---|---|
| 2 years | $24 \cdot 745{,}50$ € |
| **Meetings** | |
| WISEBED Technical Meeting | 736.03 € |
| FRONTS 3rd Unifying Experiment Workshop | 563.00 € |
| FRONTS 2nd Unifying Experiment Workshop | 329.48 € |
| FRONTS 1st Unifying Exp. Workshop | 302.10 € |
| WISEBED Programming Week | 371.70 € |
| FRONTS 2nd Winter School | 350.00 € |
| **Conferences** | |
| SESENA Workshop | 826.76 € |
| | |
| **Total** | 21,371.07 € |

**Table 5.2:** Expenses in two years of work

# Chapter 6

# Conclusions

In this final chapter I will comment my conclusions, starting with the time synchronization and topology algorithms, then with the Wisebed project and the wireless sensor networks as a whole, and finally about this project itself.

## 6.1 Time synchronization

A comparison of the algorithms implemented was done in section 3.8, in which the technical details are compared and discussed. As of time synchronization algorithms in general, they are with no doubt very important in wireless sensor networks, not only because synchronization is useful in many situations, but because sensors' clocks are very inaccurate and their error grows very rapidly.

Three algorithms were implemented and tested (LTS, TPSN and HRTS), LTS is a simple sender-receiver algorithms, TPSN is superior to LTS in number of messages sent and fault tolerance, and HRTS has less error for being a receiver-receiver algorithm. Between this three algorithms, the one that is better prepared to work on the real world is TPSN, as it is the only one that offer fault tolerance.

The error of the clocks after synchronization has an order of hundreds of microseconds at one hop distance, so in a whole network we can expect an error of milliseconds. This is the best accuracy possible if using clocks such as the one on iSense, that gives time with millisecond precision. If better synchronization is required, by applications wanting to measure distances from sound, for example, more precise clocks would be needed, but the algorithms used could be the same.

## 6.2 Topology

Topology algorithms implemented in Wiselib were compared in section 4.5. There it was commented that LMST and FLSS are the best choices if minimal energy topologies are required (and position information available), KNEIGH and XTC are better if simpler algorithms are desired, and CBTC only if a directional antenna can be used and there is no positioning system.

We can note that XTC algorithm is the only one relying on the concept of link quality: it weights edges on the network by their link quality, and not by their distance. In networks without obstacles, the results by LMST will have a less energy consuming topology than XTC, but in a network deployed in a building or similar, the XTC algorithm will build a much more realistic topology based on energy distance better than euclidean distance.

It is important to note that none of these algorithms have fault tolerance inherent to their algorithm, for example by defining the behaviour if no answer is received by a neighbor during the running of the algorithm. Their solution is to run the algorithm periodically, thus applying any change in the network distribution.

## 6.3 Wisebed project

The Wisebed project, three years long, has its final review the same day I am delivering this memory. As part of it for the last two years, I have seen great progress in the wireless sensor networks field and most of the goals accomplished.

The part of the project in which I have been more active is the Wiselib library implementation. This library has grown in number of algorithms (dozens implemented in many different fields), in the number of platforms and specially in their supported interfaces.

Not only it has grown in size, but also in usability and coherence. A lot of effort was put to be able to implement generic applications that work on any platform, and lots of re-factoring had to be made on summer 2010 in order to build this applications, support many algorithms working at the same time, and covering the requirements of FRONTS project.

The Wiselib development has not been either a pure implementation challenge, as it required deep thought of each algorithm type, defining common interfaces, requirements and dependencies for each of them. A proof of this is that an algorithm not only has its implementation in classes, but its relation to a concept and a model, which are only part of the documentation, but help understanding the work of each algorithm.

A goal that was not accomplished was to move all the algorithms from the testing to the stable branch of the library by the end of the project. This was later found impossible, as it meant testing every algorithm on all the different platforms, consuming lots of time and resources. In spite of this, most of the goals were covered, and in fact some of them, as the number of algorithms implemented, is greater than the one expected.

A proof that the Wisebed is working in all of its aspects: software development and testing, is that the FRONTS project has been implemented into the Wiselib and has used some of the testbeds with great results. It is very important for the Wisebed project to have this relation to other projects, as it helps it to improve and adapt to real necessities.

## 6.4 Wireless sensor networks

Working on the wireless sensor networks field, it has been made clear to me the real gap between academic research and real implementation of sensors. While lots of algorithms are published improving theoretical properties, sensors are manufactured by companies that implement their own closed source algorithms in a platform dependent way, without publishing any results on tests or implementation details.

WSN is indeed an active field of research, and the own existence of FRONTS and Wisebed projects is a sample of it. Algorithms are developed every year, sensors are acquired by universities and tests are performed, and collaborative development platforms like Wiselib have been created. All this advances are already making WSN a more accessible field of research, and practical results will be seen soon.

It is also very important the task in project as Wisebed of forming students, PhD. students and professors in the use and development on WSN. The Wisebed project for example has tens of PhD. students working on it that will in their future keep developing and improving algorithms for these networks. I had a very friendly environment both in UPC and in the European meetings, which in my opinion helps a lot the proper development of big projects.

## 6.5 My project

My work on this field: the development and test of some algorithms, has been a very challenging and rewarding work. As part of the development of a free software library, it will be accessible and ready for change and improvement by any researcher. I dived into this field without knowing about the existence of WSN, and by studying algorithms, implementing them and being in touch with many partners of the project, have learned a lot about the field, and also about collaborative research and development. It has been a very formative work and I'm proud for having been part of the UPC in this European project.

# Appendix A

# Wiselib development

We have commented previously that the Wiselib is implemented in C++, making an extensive use of templates. In this chapter we will take a closer look at the Wiselib implementation. We will see an example of each of the Wiselib parts:

1. **Internal interface**: data structures and auxiliary classes.
2. **External interface**: platform-dependent code to provide a common interface.
3. **Algorithms**: algorithms implemented in a platform-independent manner.
4. **Applications**: generic applications that will work on any platform, and which can use algorithms.

## A.1 Previous matters

This section will introduce two important building blocks of the Wiselib.

### A.1.1 Template-based design

The Wiselib uses an Object Oriented design based only on templates. It does not achieve generality by deriving from common interfaces, for which `vtables` would be necessary, meaning a runtime and memory footprint due to pointer indirection. Every class gets the required interfaces implementation through template parameters. For example, a topology algorithm is templated as:

```
template< typename ...,
          typename Radio,
          ... >
class LmstTopology
{ ... };
```

**Listing 27: Templated class** - template header of an algorithm class.

And when it is instantiated from an application, we use a code such as:

```
LmstTopology< ... ,
              ExtendedRadio
              ...>
   my_topology;
```

**Listing 28: Templated class instantiation** - instantiation of an algorithm class.

This way the `ExtendedRadio` can derive from other classes or simply implement the defined interface, and the topology algorithm can benefit of compiler optimizations and function inlining.

### A.1.2 Callbacks and delegates

It is very important for a library such as the Wiselib to support callback functions. Classes need to be able to register their methods in other classes. As mentioned before, the Wiselib design refuses to use the common approach of virtual inheritance. Instead, the classes need to be able to register any other method of another class as a callback, provided the method signature is correct. For example, the DBFS algorithm provides the following registration method, which will be called when the algorithm finishes its execution:

```
template< ... >
class DbfsGraph
{
   ...
   template< typename T,
             void (T::*TMethod)() >
   inline void reg_finish_callback( T *obj_pnt )
   { ... };
   ...
};
```

**Listing 29: Register callback** - callback registration function in an algorithm.

To implement these functions, we use the extremely efficient delegate implementation by Sergey Ryazanov (23). A delegate behaves as a pointer to a specific of a specific object or class.

### A.1.3 Wiselib structure

The Wiselib library is organized with three main branches: stable, testing and incubation. Stable algorithms have been tested on all platforms and have proved its correctness. Programs in testing have been tested on some platform, but need still to be tested on the rest to be able to move to stable. Incubation algorithms, instead, are programmed in a way that they will never work on all platforms.

In the Wiselib trunk, at the Subversion repository (33), four folders have our interest:

1. `applications`: generic Wiselib applications, platform-independent.
2. `wiselib.stable`: stable repository.
3. `wiselib.testing`: testing repository.
4. `wiselib.incubation`: incubation repository.

And inside the `wiselib.stable`, `wiselib.testing` and `wiselib.incubation` branches, we find the same structure:

1. `internal_interface`: contains data structures and other concepts, such as `position`, `routing_table`, `position`...

2. `external_interface`: platform-dependent code to provide common interfaces. Organized with a folder for each platform: `isense`, `lorien`, `osa`, `scw`, `shawn`, `tinyos`...

3. `algorithms`: contains all the algorithm, organized by folders representing the category of the algorithm: `cluster`, `coloring`, `crypto`, `graph`, `localization`, `routing`...

## A.2 Internal interface

The internal interface, organized as we can see in the Figure A.1, provides a set of data structures. The Wiselib algorithms need to work on many different platforms. These includes platforms based on C, and others with uncommon C++ compilers which do not support the Standard Template Library (STL). For this reason it is necessary to implement our own data structures in a efficient way, and with restrictions such as static memory allocation.

As an example, we can see the priority queue implementation in the Code 30. This class was implemented by myself because it was needed for Prim's algorithm in LMST topology algorithm.

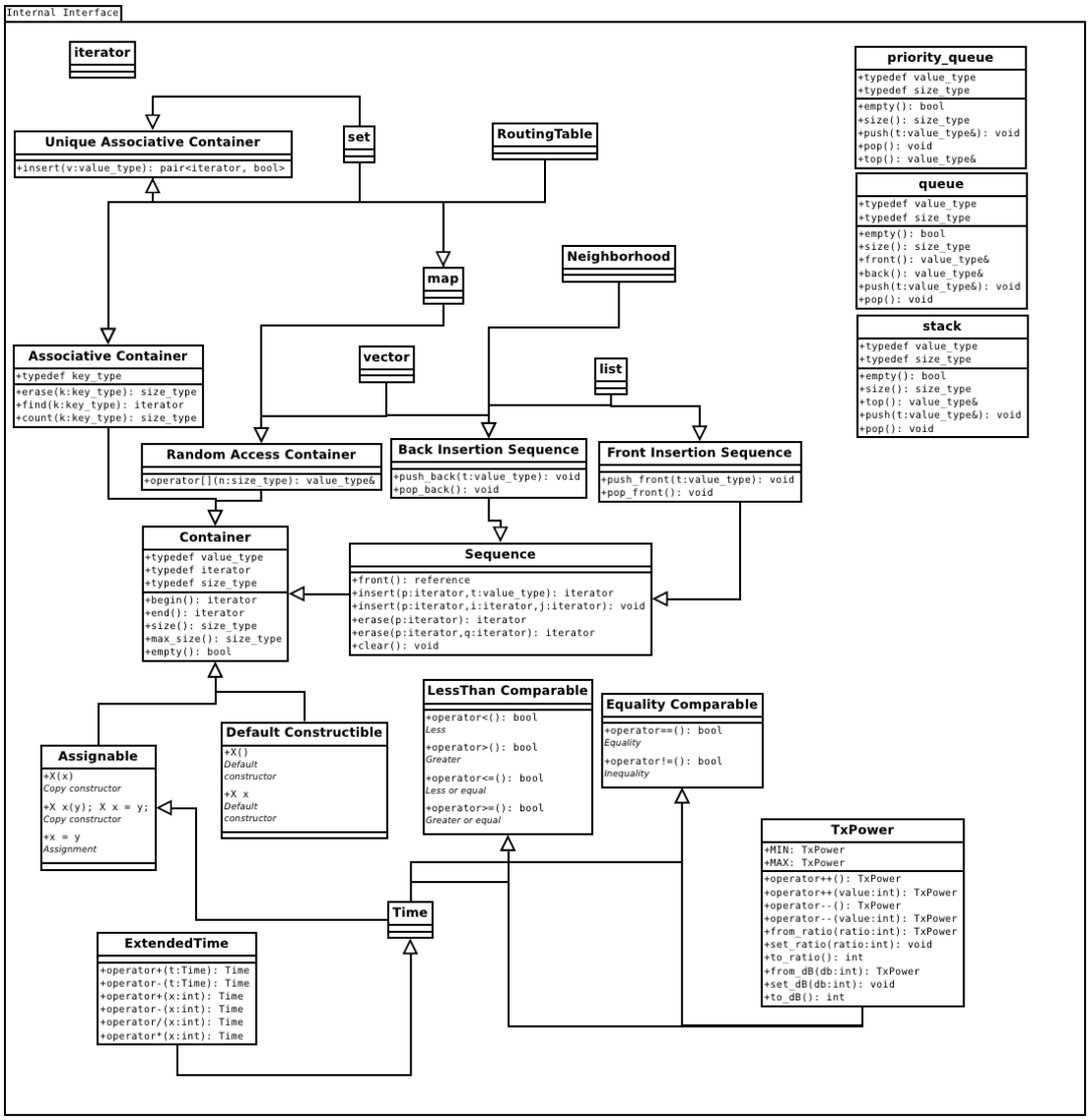



**Figure A.1: Internal interface diagram** - Internal interface concepts diagram

```
template<typename OsModel_P,
         typename Value_P,
         int QUEUE_SIZE>
class priority_queue
{
public:
   typedef Value_P value_type;
   typedef value_type* pointer;

   typedef typename OsModel_P::size_t size_type;

   priority_queue()
   {
      start_ = &vec_[0];
      finish_ = start_;
      end_of_storage_ = start_ + QUEUE_SIZE;
   }
   ...
   size_type size() { ... }
   ...
   void push( const value_type& x ) { ... }
   ...
private:
   value_type vec_[QUEUE_SIZE];
   pointer start_, finish_, end_of_storage_;
};
```

**Listing 30: Priority queue implementation** - `svn:/wiselib.testing/util/pstl/priority_queue.h`.

## A.3 External interface

The external interface, as mentioned before, provides interfaces for different platform-dependent calls. Given the system calls for sending messages, getting the position, setting a timer... the external interface programs give a common interface so algorithms can be programmed in a platform independent fashion. In Figure A.2 we can find the

concept hierarchy for the external interface.

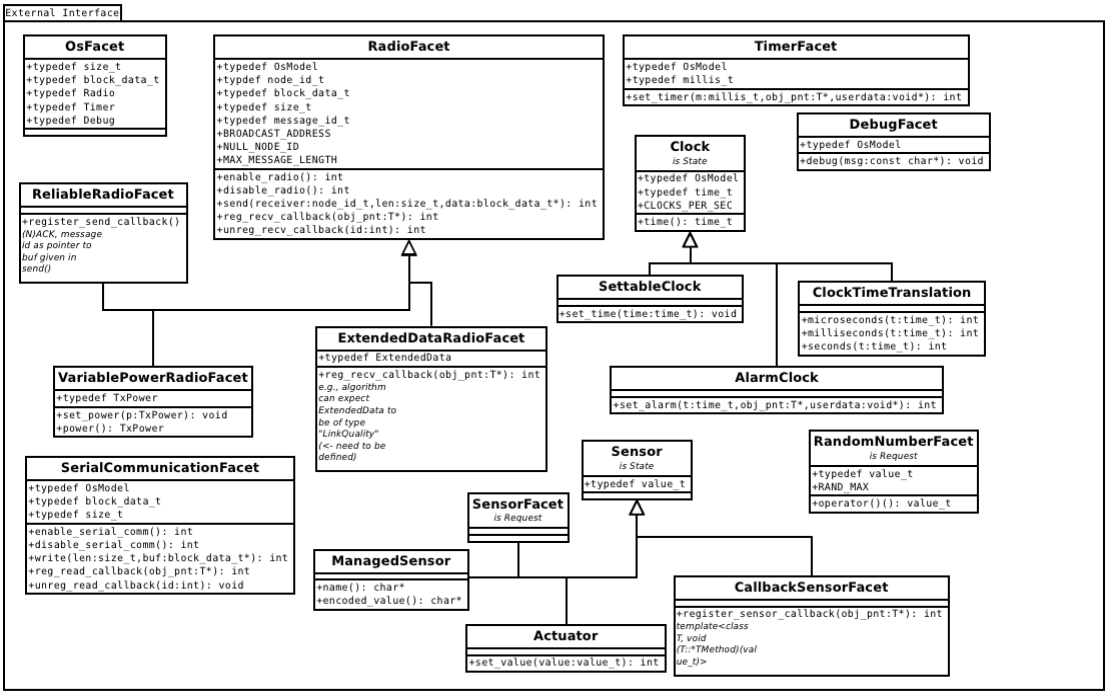



**Figure A.2: External interface diagram** - External interface concepts diagram

As an example, we can see the clock interface for the shawn simulator, which was also implemented by myself. It is important to note how it implements the functions specified in diagram A.2 `time()`, `microseconds(time)`, `milliseconds(time)` and `seconds(time)`, using specific shawn calls.

```
template<typename OsModel_P>
class ShawnClockModel
{
public:
   typedef OsModel_P OsModel;
   ...
   typedef double time_t;
   ...

   time_t time()
   {
      return os().proc->owner().world().current_time();
   }

   uint16_t microseconds( time_t time )
   { return 0; }

   uint16_t milliseconds( time_t time )
   { return (uint16_t)(time - int(time)) * 1000; }

   uint32_t seconds( time_t time )
   { return (uint32_t)time; }

private:
   ShawnOs& os()
   { return os_; }

   ShawnOs& os_;
};
```

**Listing 31: Shawn clock implementation** - svn:/wiselib.testing/external_interface/shawn/shawn_clock.h.

## A.4 Algorithms

In the algorithm folder (one in `wiselib.stable`, one in `wiselib.testing` and one in `wiselib.incubation` we can find the implementation of all the algorithms supported. They are grouped in subfolders, as mentioned earlier. They are also organized logically as in the Figure A.3.

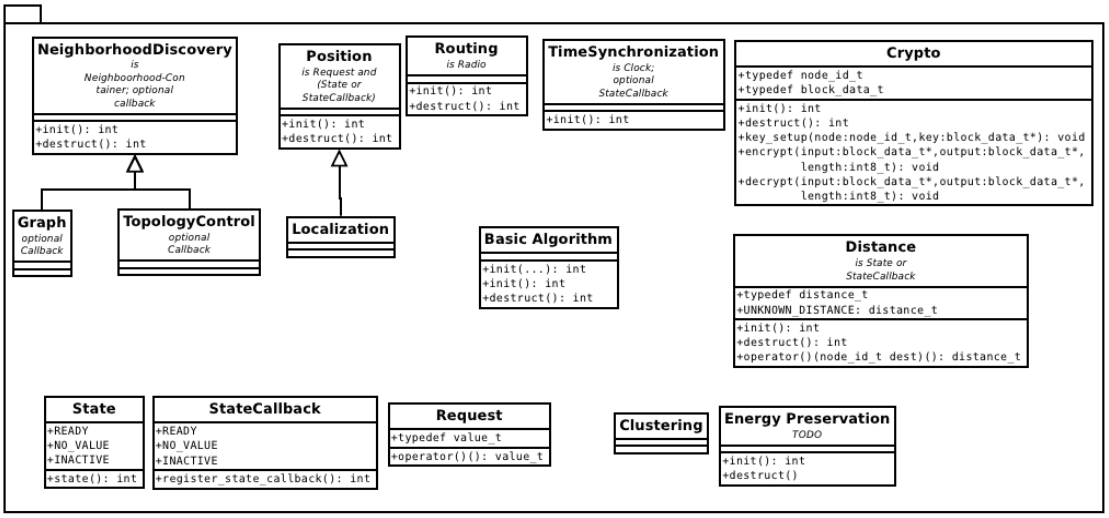



**Figure A.3: Algorithms diagram** - Algorithm concepts diagram

Algorithms implementation will not be discussed here, as it is a matter widely explained in chapters 4 and 3.

## A.5 Applications

The Wiselib also provides the possibility of easily developing applications which are platform independent, and which can use any of the data structures, interfaces and algorithms the library contains.

In the next example we can see a whole applications which starts the LTS synchronization algorithm in every node.

```
#include "external_interface/external_interface.h"
#include "external_interface/external_interface_testing.h"
#include "algorithms/synchronization/lts/lts_synchronization.h"

typedef wiselib::OSMODEL Os;

typedef wiselib::LtsSynchronization<Os> lts_synchronization_t;

class LtsSynchronizationApplication
{
   public:
      void init( Os::AppMainParameter& value )
      {
         Os::Radio::self_pointer_t radio =
            &wiselib::FacetProvider<Os, Os::Radio>::get_facet( value );
         Os::Timer::self_pointer_t timer =
            &wiselib::FacetProvider<Os, Os::Timer>::get_facet( value );
         Os::Clock::self_pointer_t clock =
            &wiselib::FacetProvider<Os, Os::Clock>::get_facet( value );
         Os::Debug::self_pointer_t debug =
            &wiselib::FacetProvider<Os, Os::Debug>::get_facet( value );

         synchronization.init( *radio, *timer, *debug, *clock );
         synchronization.enable();
      }
   private:
      lts_synchronization_t synchronization;
};

wiselib::WiselibApplication<Os, LtsSynchronizationApplication>
   synchronization_app;

void application_main( Os::AppMainParameter& value )
{
   synchronization_app.init( value );
}
```

**Listing 32: Synchronization application** - `svn:/applications/sync_test/sync_test.cpp`.

This application can be compiled for any supported platform, as everything is done depending on the `OSMODEL` parameter, and it has the common interface `application_main`. The Wiselib has Makefiles which when modified with the correct paths, have the necessary rules to compile these applications for any kind of sensor and simulator.

# Appendix B

# Simulation with Shawn

## B.1 Introduction

It has been commented many times through the document that Shawn simulator (24) is used for simulation the behaviour of the algorithms, and checking its correct execution.

We will see in this appendix how an application is compiled and simulated with Shawn, what options we can configure in the simulator, how to interpret the output and visualize the result.

The DBFS algorithm is chosen for this example simulation, as it is a simple algorithm, and its result can be visualized easily.

## B.2 Application

It is commented in Section A.5 how an example application is programmed. In the first stages of the Wiselib it was necessary to dive into Shawn code and build a Shawn application in order to test the algorithms. Later on, generic Wiselib applications were made possible, and so it is much easier to test algorithms in Shawn. We see in Program code 33 the application used in this case.

```
#include "external_interface/external_interface.h"
#include "algorithms/graph/dbfs/dbfs_graph.h"

typedef wiselib::OSMODEL Os;
```

```
typedef wiselib::DbfsGraph<Os> graph_t;

class GraphApplication
{
   public:
      typedef Os::Position::position_t tipus_posicio;
      void init( Os::AppMainParameter& value )
      {
         radio = &wiselib::FacetProvider<Os, Os::Radio>::get_facet(value);
         timer = &wiselib::FacetProvider<Os, Os::Timer>::get_facet(value);
         debug = &wiselib::FacetProvider<Os, Os::Debug>::get_facet(value);

         graph.init( *radio, *timer, *debug );
                        if (radio->id() == 0)
                              graph.set_root();
         graph.enable();
      }

   private:
      graph_t graph;
      Os::Radio *radio;
      Os::Timer *timer;
      Os::Debug *debug;
};

// --------------------------------------------------------

wiselib::WiselibApplication<Os, GraphApplication> graph_app;
void application_main( Os::AppMainParameter& value )
{
   graph_app.init( value );
}
```

**Listing 33: DBFS example application** - `graph_test.cpp`, application to be simulated in Shawn

Note that before enabling the algorithm, its operating system facets: radio, timer and debug must be initialized. Most of the code in this application is part of any generic application, and the important call is `graph.enable();`.

```
all: shawn

export APP_SRC=graph_test.cpp
export BIN_OUT=graph_test

 include ../Makefile
```

**Listing 34: Application Makefile** - includes the general Wiselib Makefile that has the Shawn compilation methods

We will compile this function by executing `make shawn`, which will call the Makefile in 34, which will use the default Wiselib Makefile (`include ../Makefile`).

## B.3 Simulation configuration

To execute the simulation, we will use the command

```
./graph_test -f shawn.conf
```

, where `shawn.conf` is a file containing the Shawn configuration. In Listing 35 we can see the Shawn configuration used in our example. In the comments we can see the explanation of each parameter.

```
# Set random seed, allows repeating simulations
random_seed action=set seed=1789

# prepare_world: prepare scenario
# edge_model=list comm_model=disk_graph: data structures used
#        to represents nodes and communication
# range=50: radio range of 50 length units
prepare_world edge_model=list comm_model=disk_graph range=500
         transm_model=stats_chain
chain_transm_model name=reliable
```

```
# rect_world width=100 height=100: rectangular 100x100 world
# count=5: 5 sensors
# processors=wiselib_shawn_standalone: load sensors with
#          wiselib application
rect_world width=1000 height=1000 count=5
          processors=wiselib_shawn_standalone

# simulation max_iterations=20: simulation will last
#          20 seconds (iterations)
simulation max_iterations=20
```

**Listing 35: Shawn configuration file** - configuration file used in the DBFS example simulation

## B.4 Execution

We can now execute the simulation, using the command commented above:

```
./graph_test -f shawn.conf
```

Following is the output of this command:

```
------------------------------ BEGIN ITERATION 0
------------------------------ DONE ITERATION 0
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 1
------------------------------ DONE ITERATION 1
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 2
------------------------------ DONE ITERATION 2
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 3
0: DbfsMsgIdNeighbourhood message from 3
0: DbfsMsgIdNeighbourhood message from 1
```

```
2: DbfsMsgIdNeighbourhood message from 1
1: DbfsMsgIdNeighbourhood message from 2
1: DbfsMsgIdNeighbourhood message from 0
3: DbfsMsgIdNeighbourhood message from 0
------------------------------ DONE ITERATION 3
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 4
------------------------------ DONE ITERATION 4
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 5
0: Executing TimerElapsed 'DbfsGraph'
------------------------------ DONE ITERATION 5
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 6
3: DbfsMsgIdLabel message from 0
1: DbfsMsgIdLabel message from 0
------------------------------ DONE ITERATION 6
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 7
0: DbfsMsgIdEcho message from 3
0: DbfsMsgIdEcho message from 1
------------------------------ DONE ITERATION 7
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 8
1: DbfsMsgIdLabel message from 0
------------------------------ DONE ITERATION 8
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 9
2: DbfsMsgIdLabel message from 1
------------------------------ DONE ITERATION 9
```

```
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 10
1: DbfsMsgIdEcho message from 2
------------------------------ DONE ITERATION 10
  [ 5 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 11
0: DbfsMsgIdEcho message from 1
0: Stop of the algorithm
------------------------------ DONE ITERATION 11
```

**Listing 36: DBFS simulation debug** - simulation output by Shawn simulator

To understand this output, it is necessary to remember the DBFS algorithm, explained in Section 3.3.3.1 on page 28.

The output in the simulation are debug messages sent by DBFS algorithm (in our application we didn't send any debug message). We can identify the next phases in this simulation:

- **Iteration 3**: Neighborhood discovery messages are sent, so every node knows who are its neighbors.
- **Itearion 5**: root (node 0), executes a timer, and will start the graph building.
- **Itearion 6**: root (node 0), labels nodes 3 and 1.
- **Itearion 7**: 3 and 1 return an Echo to 0.
- **Itearion 8**: 0 sends a label message to 1. 1 must have other neighbors to be explored.
- **Itearion 9**: 1 labels 2.
- **Itearion 10**: 2 returns an Echo to 1.
- **Itearion 10**: 1 return the Echo to 0. The algorithm stops.

From this trace we can deduce the graph has the form: $0 \rightarrow 3$, $0 \rightarrow 1$ and $1 \rightarrow 2$. We have seen through this simulation that the messages are sent in the expected order, and the algorithm behaves as it should. Still, it would be desirable to visualize the results.

## B.5 Visualization

We will use the visualizer written in Python by Antoni. This algorithm reads text input files, and parses lines that describe nodes and edges. These lines have the following format:

- `@ POS_X # POS_Y # ID # IS_ROOT` , description of node `ID`.
- `$ A -> B $ TYPE` , edge from `A` to `B`. `TYPE` specifies if the link is in the graph (`h`), or not (`c`).

We add the function in Program code 37 to the application, which will be run after the graph is built. This function outputs for every node its position, its neighbors and its parent.

```
void draw( void* )
{
   tipus_posicio pos = position->position();
   debug->debug("@%f#%f#%d#1#%d#0\n", pos.x(), pos.y(), radio->id(), 0 == radio->id());
   for (int i  = 0; i < graph.neighbors_.size(); ++i)
      debug->debug("$%d->%d$c\n", radio->id(), graph.neighbors_[i]);
   debug->debug("$%d->%d$h\n", radio->id(), graph.parent_);
}
```

**Listing 37: Draw function** - writes the input needed by the visualizer to draw the graph

Now we simply have to store the output of the simulation in a file, and call the visualizer:

```
./graph_test -f shawn.conf &> draw.txt
./visor/visor.py -b -y draw.txt
```

Options `-b` and `-y` write the root of the tree bigger, and paints the graph edges yellow. Finally, the image of the resulting graph is in Figure B.1, and we check the graph is the one we deduced before ($0 \to 3$, $0 \to 1$ and $1 \to 2$).

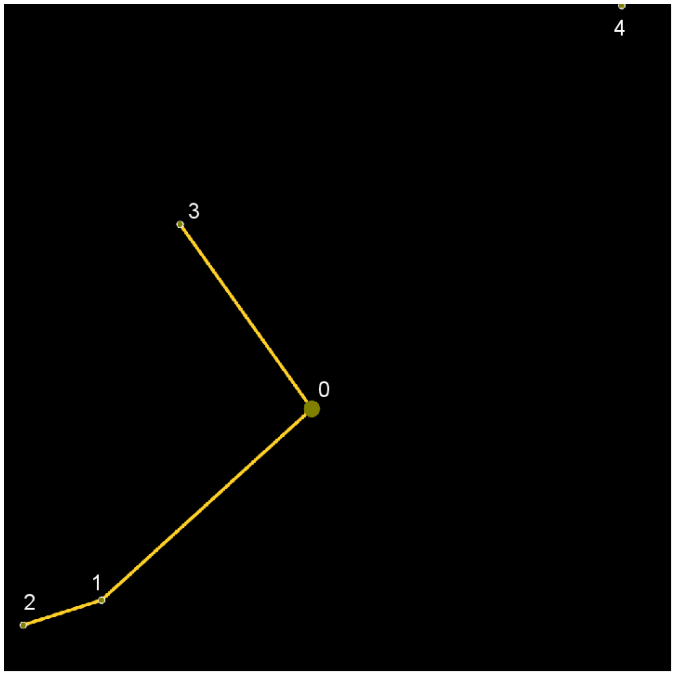



**Figure B.1: Visualization of DBFS simulation** - result of the simulation of DBFS with 5 nodes

# Appendix C

# Program Codes

In this appendix can be found the most important functions of the algorithms, cited through the rest of the document.

## C.1 DDFS

```
template<typename OsModel_P,
         typename Radio_P,
         typename Debug_P,
         uint16_t MAX_NODES>
void
DdfsGraph<OsModel_P, Radio_P, Debug_P, MAX_NODES>::
receive( node_id_t from, size_t len, block_data_t *data )
{
   uint8_t msg_id = *data;
   if ( msg_id == DdfsMsgIdNeighbourhood and from != radio().id() )
   {
      neighbours_.push_back( from );
      unvisited_.push_back( from );
      flag_.push_back( false );
   }
   else if ( msg_id == DdfsMsgIdDiscover )
   { // I'm visited for the first time
      parent_ = from;
      for ( int i = 0; i < (int)neighbours_.size(); ++i )
```

```
    {
       message_ = DdfsMsgIdVisited;
       radio().send( neighbours_[i], 1, (uint8_t*)&message_ );
       flag_[i] = true;
    }
    if ( neighbours_.size() == 1 and neighbours_[0] == from )
    { // from is my only neighbor
       message_ = DdfsMsgIdReturn;
       if ( parent_ == radio().id() )
          receive( radio().id(), 1, (uint8_t*)&message_ );
       else
          radio().send( parent_, 1, (uint8_t*)&message_ );
    }
 }
 else if ( msg_id == DdfsMsgIdReturn )
 { // the search is resumed from me,
   //which I've already been visited
    if ( from != radio().id() )
       children_.push_back( from );
    if ( not unvisited_.empty() )
    {
       message_ = DdfsMsgIdDiscover;
       radio().send( unvisited_[unvisited_.size() - 1],
                     1, (uint8_t*)&message_ );
       unvisited_.pop_back();
    }
    else // all neighbours are visited
    {
       if ( parent_ != radio().id() )
       {
          message_ = DdfsMsgIdReturn;
          if ( parent_ == radio().id() )
             receive( radio().id(), 1, (uint8_t*)&message_ );
          else
             radio().send( parent_, 1, (uint8_t*)&message_ );
       }
```

```
            else
            {
               // STOP of the algorithm
               if ( set_ddfs_delegate_ )
                  ddfs_delegate_();
            }
         }
      }
      else if ( msg_id == DdfsMsgIdVisited )
      {
         int erase_position = -1;
         for ( int i = 0; i < (int)unvisited_.size(); ++i )
            if ( unvisited_[i] == from )
               erase_position = i;
         if ( erase_position != -1 )
            unvisited_.erase(unvisited_.begin() + erase_position);
         message_ = DdfsMsgIdAck;
         radio().send( from, 1, (uint8_t*)&message_ );
      }
      else if ( msg_id == DdfsMsgIdAck )
      {
         for ( int i = 0; i < (int)neighbours_.size(); ++i )
            if ( neighbours_[i] == from )
               flag_[i] = false;
         bool all_false = true;
         for ( int i = 0; all_false and i < neighbours_.size(); ++i )
            if ( flag_[i] == true )
               all_false = false;
         if ( all_false )
         {
            message_ = DdfsMsgIdReturn;
            receive( radio().id(), 1, (uint8_t*)&message_ );
         }
      }
   }
```

**Listing 38: DDFS receive** - handles received messages with the radio

## C.2 DBFS

```
template<typename OsModel_P,
         typename Radio_P,
         typename Debug_P,
         uint16_t MAX_NODES>
void
DbfsGraph<OsModel_P, Radio_P, Debug_P, MAX_NODES>::
receive( node_id_t from, size_t len, block_data_t *data )
{
   uint8_t msg_id = *data;
   if ( msg_id == DbfsMsgIdNeighbourhood and from != radio().id() )
   {
      neighbours_.push_back( from );
   }
   else if ( msg_id == DbfsMsgIdLabel )
   { // I'm visited for the first time
      if (labeled_ == false) {
         labeled_ = true;
         parent_ = from;
         ++data;
         level_ = *data + 1;
         send_to_ = neighbours_;
         echoed_.clear();
         for (int i = 0; i < (int)send_to_.size(); ++i)
            echoed_.push_back(false);
         for (int i = 0; i < (int)send_to_.size(); ++i)
            if (send_to_[i] == from) {
               send_to_.erase(send_to_.begin() + i);
               echoed_.erase(echoed_.begin() + i);
            }
         children_.clear();
         message_[0] = DbfsMsgIdEcho;
         if (send_to_.empty())
```

```
            message_[1] = EchoEnd;
         else
            message_[1] = EchoKeepon;
         radio().send( parent_, 2, (uint8_t*)&message_ );
      }
      else {
         if (parent_ == from) {
            for (int i = 0; i < (int)send_to_.size(); ++i) {
               message_[0] = DbfsMsgIdLabel;
               message_[1] = level_;
               radio().send( send_to_[i], 2, (uint8_t*)&message_ );
               echoed_[i] = false;
            }
         }
         else {
            message_[0] = DbfsMsgIdEcho;
            message_[1] = EchoStop;
            radio().send( from, 2, (uint8_t*)&message_ );
         }
      }
   }
   else if ( msg_id == DbfsMsgIdEcho )
   { // the search is resumed from me
     //, which I've already been visited
      for (int i = 0; i < (int)send_to_.size(); ++i)
         if (send_to_[i] == from)
            echoed_[i] = true;
      uint8_t status = *(++data);
      if (status == EchoKeepon) {
         bool is_children = false;
         for (int i = 0; i < children_.size() and not is_children; ++i)
            if (children_[i] == from)
               is_children = true;
         if (not is_children)
            children_.push_back(from);
      }
```

```
    else if (status == EchoStop) {
        for (int i = 0; i < (int)send_to_.size(); ++i)
            if (send_to_[i] == from) {
                send_to_.erase(send_to_.begin() + i);
                echoed_.erase(echoed_.begin() + i);
            }
    }
    else if (status == EchoEnd) {
        bool is_children = false;
        for (int i = 0; i < children_.size() and not is_children; ++i)
            if (children_[i] == from)
                is_children = true;
        if (not is_children)
            children_.push_back(from);
        for (int i = 0; i < (int)send_to_.size(); ++i)
            if (send_to_[i] == from) {
                send_to_.erase(send_to_.begin() + i);
                echoed_.erase(echoed_.begin() + i);
            }
    }
    if (send_to_.empty()) {
        if (root_) {
            if ( set_dbfs_delegate_ )
                dbfs_delegate_();
        }
        else {
            message_[0] = DbfsMsgIdEcho;
            message_[1] = EchoEnd;
            radio().send( parent_, 2, (uint8_t*)&message_ );
        }
    }
    else {
        bool all_echoed = true;
        for (int i = 0; i < (int)echoed_.size() and all_echoed; ++i)
            if (not echoed_[i])
                all_echoed = false;
```

```
            if (all_echoed) {
               if (root_)
                  for (int i = 0; i < (int)send_to_.size(); ++i) {
                     message_[0] = DbfsMsgIdLabel;
                     message_[1] = level_;
                     radio().send( send_to_[i], 2, (uint8_t*)&message_ );
                     echoed_[i] = false;
                  }
               else {
                  message_[0] = DbfsMsgIdEcho;
                  message_[1] = EchoKeepon;
                  radio().send( parent_, 2, (uint8_t*)&message_ );
               }
            }
         }
      }
   }
```

**Listing 39: DBFS receive()** - `receive()` function, most important in DBFS algorithm

## C.3 LTS

```
receive( node_id_t from, size_t len, block_data_t *data )
{
   uint8_t msg_id = *data;
   if ( msg_id == LtsMsgIdSynchronizationPulse )
   {
      synchronizationMessage.set_t2( clock().time() );
      SynchronizationMessage *msg = ( SynchronizationMessage * )data;
      synchronizationMessage.set_msg_id( LtsMsgIdAcknowledgement );
      synchronizationMessage.set_t1( msg->t1() );
      synchronizationMessage.set_t3( clock().time() );
      radio().send( from, 1 + 3*TIME_SIZE,
                    (uint8_t*)&synchronizationMessage );
   }
   else if ( msg_id == LtsMsgIdAcknowledgement )
```

```
   {
      Time t4 = clock().time();
      SynchronizationMessage *msg = (SynchronizationMessage *)data;
      Time offset = ( msg->t2() + msg->t3() - msg->t1() - t4  )/2;
      synchronizationMessage.set_msg_id( LtsMsgIdOffset );
      synchronizationMessage.set_t1( offset );
      radio().send( from, 1 + TIME_SIZE,
                    (uint8_t*)&synchronizationMessage );
   }
   else if ( msg_id == LtsMsgIdOffset ) {
      SynchronizationMessage *msg = (SynchronizationMessage *)data;
      clock().set_time( clock().time() + msg->t1() );
      notify_listeners();
      start_synchronization();
   }
}
```

**Listing 40: LTS receive** - receive function

## C.4 TPSN

```
timer_elapsed( void* userdata )
{
   if ( level_ == 0 )
   {
      if ( not built_tree_ )
      {
         // I'm root and I'll start the tree construction
         built_tree_ = true;
         levelDiscoveryMessage[1] = level_;
         radio().send( radio().BROADCAST_ADDRESS, 2,
                       (uint8_t*)&levelDiscoveryMessage );
         timer().template set_timer<self_type,
                          &self_type::timer_elapsed>(
                          tree_construction_time_, this, 0 );
      }
```

```
    else
    {
      // I'm root and I'll start the synchronization
      radio().send( radio().BROADCAST_ADDRESS, 1,
                    (uint8_t*)&timeSyncMessage );
    }
  }
  else
  {
    if ( level_ == -1 )
    {
      // I'm not root and I don't have a father
      if ( new_level_ == -1 )
      {
        // I neither have found a new father yet,
        // I request one
        requested_father_ = true;

        radio().send( radio().BROADCAST_ADDRESS, 1,
                      (uint8_t*)&levelRequestMessage );
        timer().template set_timer<self_type,
                        &self_type::timer_elapsed>(
                        timeout_, this, 0 );
      }
      else
      {
        // I have a new father and request synchronization
        level_ = new_level_;
        requested_father_ = false;
        send_sync_pulse();
      }
    }
    else if ( not synchronized_ )
    {
      // I'm not root and I have a father
      // but I'm not synchronized
```

```
        if ( retries_ < MAX_RETRIES )
        {
           // I request synchronization
           ++retries_;
           send_sync_pulse();
        }
        else
        {
           // I tried synchronizing 4 times,
           // and now request a father
           retries_ = 0;
           level_ = -1;
           new_level_ = -1;
           requested_father_ = true;
           radio().send( radio().BROADCAST_ADDRESS, 1,
                         (uint8_t*)&levelRequestMessage );
           timer().template set_timer<self_type,
              &self_type::timer_elapsed>( timeout_, this, 0 );
        }
     }
  }
}
```

**Listing 41: TPSN timer elapsed** - function called after different timeouts

## C.5 HRTS

```
receive( node_id_t from, size_t len, block_data_t *data )
{
   time_t t = clock().time();
   if ( from == radio().id() )
      return;
   uint8_t msg_id = *data;
   message = *((Synchronization *) data);
   if ( msg_id == HrtsMsgIdBeginSync )
   {
```

```
      uint8_t level = *data;
      if ( level_ != -1 and level_ > message.level )
         return;
      t1 = message.t1();
      if (message.receiver == radio().id()) {
         message.set_msg_id(HrtsMsgIdReply);
         message.set_t1(t);
         message.set_t2(clock().time());
         radio().send( from, sizeof( SynchronizationMessage ),
                       (uint8_t*)&message );
      }
   }
   else if ( msg_id == HrtsMsgIdReply )
   {
      time_t d2 = ((message.t1() - t1) - (t - message.t2()))/2;
      message.set_msg_id(HrtsMsgIdDiffSync);
      message.set_t1(message.t1());
      message.set_t2(d2);
      radio().send( radio().BROADCAST_ADDRESS,
                    sizeof( SynchronizationMessage ), (uint8_t*)&message );
   }
   else if ( msg_id == HrtsMsgIdDiffSync )
   {
      clock().set_time(clock_time()+message.t2()+message.t1()-t1);
   }
}
```

**Listing 42: HRTS receive** - receive function for HRTS

## C.6 LMST

```
generate_topology()
{
   // Prim's algorithm to find the MST of the
   // visible neighborhood graph
   node_id_t me = NV.size();
```

```
NV.push_back( radio().id() );
Pos.push_back( loc_->position() );
D.clear();
for ( size_t i = 0; i < NV.size(); ++i )
   D.push_back( std::numeric_limits<position_t>::infinity() );
p.clear();
for ( size_t i = 0; i < NV.size(); ++i )
   p.push_back( -1 );
is_in_PQ.clear();
for ( size_t i = 0; i < NV.size(); ++i )
   is_in_PQ.push_back( true );
PQ.clear();
PQ.push( PPI( 0.0, me ) );
D[me] = 0.0;
N.clear(); // we'll keep the visible neighbors
           // that have 'me' as parent
while ( not PQ.empty() )
{
   node_id_t u = PQ.top().second;
   if ( p[u] == me )
      N.push_back( NV[u] );
   PQ.pop();
   is_in_PQ[u] = false;
   for ( size_t i = 0; i < NV.size(); ++i )
   {
      position_t w = dist(Pos[i], Pos[u]);
      if ( is_in_PQ[i] && w < D[i] )
      {
         p[i] = u;
         D[i] = w;
         PQ.push( PPI(w, i) );
      }
   }
}
radius = 0.0;
for ( size_t i = 0; i < N.size(); ++i )
```

```
      if ( dist(Pos[i], Pos[me]) > radius )
         radius = dist( Pos[i], Pos[me] );
   NV.clear();
   Pos.clear();
}
```

**Listing 43: LMST generate topology** - Implementation of Prim's algorithm

## C.7 FLSS

```
generate_topology()
{
   // Generalized Kruskal algorithm to get the
   //    minimum spanning K-connected graph
   // Ford-fulkerson algorithm used to check K-connection
   E.clear();
   NV.push_back( radio().id() );
   n = NV.size();
   // Generate graph and fill PQ with edges
   for (int i = 0; i < n; ++i)
      for (int j = i+1; j < n; ++j) {
         capacity[2*i][2*i+1] = 1;
         capacity[2*i+1][2*j] = 0;
         capacity[2*j+1][2*i] = 0;
         E.push(pair<float_t, PII >(dist(Pos[i], Pos[j]),
                                    PII(2*i+1,  2*j)));
      }
   for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
         if (capacity[i][j] > 0)
            G[i].push_back(j);
   G_bak = G;
   capacity_bak = capacity;
   for (int i = 0; i < E.size(); ++i) {
      if (max_flow(E.top().second.first, E.top().second.second) < K) {
         G[E.top().second.first][E.top().second.second] = true;
```

```
            G[E.top().second.second+1][E.top().second.first-1] = true;
         }
         bool finish = true;
         for (int i = 0; i < n-1; ++i)
            if (max_flow(2*i+1, 2*n-1) < K)
               finish = false;
         if (finish)
            break;
      }
      N.clear();
      for (int i = 0; i < n-1; ++i)
         if (G[2*i+1][2*n-2])
            N.push_back(i);
      NV.clear();
      Pos.clear();
   }

   max_flow(node_id_t source, node_id_t sink)
   {
      G = G_bak;
      capacity = capacity_bak;
      int16_t sol = 0;
      while (true) {
         int16_t path_capacity = find_path_bfs(source, sink);
         if (path_capacity == 0)
            break;
         else
            sol += path_capacity;
      }
      return sol;
   }

   find_path_bfs(node_id_t source, node_id_t sink)
   {
      for (int i = 0; i < n; ++i)
         seen[i] = false;
```

```
for (int i = 0; i < n; ++i)
   from[i] = -1;
Q = std::queue<node_id_t>();
Q.push(source);
seen[source] = true;
while (not Q.empty()) {
   node_id_t where = Q.front();
   Q.pop();
   bool finish = false;
   for (int i = 0; i < G[where].size(); ++i)
      if ( not seen[G[where][i]] and
            capacity[where][G[where][i]] > 0 ) {
      Q.push(G[where][i]);
      seen[G[where][i]] = true;
      from[G[where][i]] = where;
      if (G[where][i] == sink) {
         finish = true;
         break;
      }
   }
   if (finish)
      break;
}
node_id_t where = sink, path_cap = inf;
while (from[where] > -1) {
   node_id_t prev = from[where];
   path_cap = (path_cap < capacity[prev][where] ?
         path_cap : capacity[prev][where]);
   where = prev;
}
where = sink;
while (from[where] > -1) {
   node_id_t prev = from[where];
   capacity[prev][where] -= path_cap;
   where = prev;
}
```

```
    if (path_cap == inf)
        return 0;
    else
        return path_cap;
}
```

**Listing 44: FLSS generate topology** - Implementation of generalized Kruskal algorithm, including Ford-Fulkerson maximum flow calculations i each iteration

# References


[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addision-Wesley, 2001. 8

[2] J. Anguera, M. Blesa, J. Farré, V. López, and J. Petit. Topology control algorithms in wiselib. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, SESENA '10, pages 14–19, New York, NY, USA, 2010. ACM. 6

[3] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147 – 150, 1985. 22

[4] T. Baumgartner, I. Chatzigiannakis, S. Fekete, C. Koninis, A. Kroller, and A. Pyrgelis. Wiselib: A generic algorithm library for heterogeneous sensor networks. In *7th European Conference on Wireless Sensor Networks*, 2010. ix, 1, 3, 7, 9, 10, 12

[5] D. Blough, M. Leoncini, G. Resta, and P. Santi. The k-neighbors protocol for symmetric topology control in ad hoc networks. In *4th ACM Interational Symposium on Mobile Ad Hoc Networking and Computing*, pages 141–152, 2003. 2, 54, 65

[6] BOOST. http://www.boost.org. 7

[7] CGAL: Computational Geometry Algorithms Library. http://www.cgal.org. 7

[8] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer. WISEBED: an open large-scale wireless sensor network testbed. In *1st Intl. Conference on Sensor Networks Applications, Experimentation and Logistics*, Lecture Notes of the Institute for Computer Sciences, Social-Inf, 2009. 1, 2, 7

[9] Coalsesenses GmbH. http://www.coaelesenses.com. 13

[10] H. Dai and R. Han. TSync: A lightweight bidirectional time synchronization service for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 8(1):125, 2004. 2, 46

[11] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507 – 518, 1975. 63

[12] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 138–149, 2003. 2, 41

[13] P. Hurni, T. Staub, G. Wagenknecht, M. Anwander, and T. Braun. A secure remote authentication, operation and management infrastructure for distributed wireless sensor network testbeds. *Electronic Communications of the EASST*, 17, 2009. 3, 16

[14] N.M. Josuttis. *The C++ Standard Library*. Addision-Wesley, 1999. 7

[15] J B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. 63

[16] LEDA - Library of Efficient Data types and Algorithms. http://www.algorithmic-solutions.com/leda/. 7

[17] LGPL. http://www.gnu.org/copyleft/lesser.html. 71

[18] L. Li, Y. Wang J. Halpern, P. Bahl, and R. Wattenhofer. Analysis of a cone-based distirbuted topology control algorithm for wireless multi-hop networks. In *20th ACM Symposium on Principles of Distributed Computing*, pages 264–273, 2001. 2, 54, 65

[19] N. Li and J. Hou. A fault-tolerant topology control algorithm for wireless sensor networks. In *10th Intl. ACM Conference on Mobile Computing and Networking*, pages 275–286, 2004. 2, 54, 61

[20] N. Li, J. C. Hou, and L. Sha. Design and analysis of an MST-based topology control algorithm. *Proc. IEEE INFOCOM 2003*, 3:1702–1712, 2003. 2, 54, 55, 59, 60

[21] L. Li P. Bahl, R. Wattenhofer and Y. Wang. Distributed topology control for power efficient operation in multihop wireless ad hoc networks. In *20th Joint Conference of the IEEE Computer and Communications Societies*, pages 1388–1397, 2001. 65

[22] R. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36:1389–1401, 1957. 55

[23] Sergey Ryazanov. http://www.codeproject.com/KB/cpp/ImpossiblyFastCppDelegate.aspx. 81

[24] S. Fischer S. Fekete, A. Kroller and D. Pfisterer. Shawn: The fast, highly customizable sensor network simulator. In *4th Intl. Conference on Networked Sensing Systems (INSS)*, page 299, 2007. 2, 3, 8, 12, 91

[25] P. Santi. *Topology control in wireless ad hoc and sensor networks*. Wiley, 2005. 19, 54

[26] P. Santi. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv.*, 37(2):164–194, 2005. 54

[27] SGLIB - A Simple Generic Library for C. http://sglib.sourceforge.net/. 7

[28] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000. 8

[29] J. v. Greunen and J. Rabaey. Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003. 2, 22, 31, 34, 37

[30] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Addision-Wesley, 2003. 8

[31] R. Wattenhofer and A. Zollinger. XTC: A practical topology control algorithm for ad hoc networks. In *18th Intl. Parallel and Distributed Processing Symposium*, 2004. 2, 54, 67

[32] WISELIB. http://www.wiselib.org. 7, 8

[33] WISELIB Subversion. https://svn.itm.uni-luebeck.de/wisebed/wiselib/trunk/. 81

[34] Feng Zhao and Leonidas J. Guibas. *Wireless sensor networks : an information processing approach*. The Morgan Kaufmann series in networking. Morgan Kaufmann, Amsterdam ; San Francisco, 2004. 2004301905 Feng Zhao, Leonidas J. Guibas. Includes bibliographical references (p. 323-345) and index. 19

[35] Yunzhou Zhu and To-Yat Cheung. A new distributed breadth-first-search algorithm. *Inf. Process. Lett.*, 25:329–334, July 1987. 22, 28